Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Auffrischung



Wintersemester 2025/2026

Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**

1.1 Fragen [5] . . . 1
1.2 Begriffe [6] . . . 1
1.3 Links zur Java-Insel [7] . . . 1

2 Typen und Operatoren **2**

2.1 Java Schlüsselworte [9] . . . 2
2.2 Kommentare [10] . . . 3
2.3 Praxis-Tipp: Kommentare [11] . . . 3
2.4 Primitive Datentypen [12] . . . 4
2.5 Literale für Ganzzahlen [13] . . . 5
2.6 Strings [14] . . . 5
2.7 StringBuilder und StringBuffer [16] . . . 6
2.8 Arrays sind Objekte [17] . . . 7
2.9 Hilfsklasse Arrays [18] . . . 8
2.10 Deklaration und Zuweisung [19] . . . 9
2.11 Arten von Operatoren [20] . . . 10
2.12 Operatoren [21] . . . 10

3 Kontrollstrukturen **12**

3.1 Kontrollstrukturen: if [23] . . . 12
3.2 Praxis-Tipp: if immer mit Block [24] . . . 12
3.3 Kontrollstrukturen: switch [25] . . . 13
3.4 Kontrollstrukturen: switch in Java 7 [26] . . . 14
3.5 Kontrollstrukturen: switch ab Java 12 [27] . . . 15
3.6 Kontrollstrukturen: for-Schleife [30] . . . 16
3.7 Kontrollstrukturen: for-each Schleife [31] . . . 17
3.8 Kontrollstrukturen: while-Schleife [32] . . . 18
3.9 Kontrollstrukturen: do/while-Schleife [33] . . . 18
3.10 break, continue und Labels [34] . . . 19
3.11 Deklaration einer Klasse [35] . . . 19

4 Speicherverwaltung **21**

4.1 Begriffe [37] . . . 21
4.2 Einheiten (alte Form) [38] . . . 21

4.3 Einheiten (neue Form) [39] . . . 22
4.4 SRAM [41] . . . 22
4.5 DRAM [42] . . . 23
4.6 Vergleich DRAM / SRAM [45] . . . 24
4.7 DRAM-für ein Byte [46] . . . 24
4.8 Speichergrid [47] . . . 25
4.9 Adresse [48] . . . 25
4.10 Bitbreiten [49] . . . 26
4.11 Speicherung eines Bytes [50] . . . 27
4.12 Größere Einheiten [51] . . . 27
4.13 Endianess [52] . . . 27
4.14 Speicherung von Objekten [54] . . . 28
4.15 Arten von Speicher [55] . . . 29
4.16 Objekt vs. Referenz [56] . . . 29

5 Quiz **31**
5.1 Quiz: Wertebereich [59] . . . 31
5.2 Quiz: Primitive Java Typen [61] . . . 31
5.3 Quiz: Kontrollstrukturen [63] . . . 32
5.4 Quiz: Arrays [67] . . . 33
5.5 Quiz: Logische Operatoren [69] . . . 33
5.6 Quiz: Parameterübergabe [71] . . . 34

Index **ii**

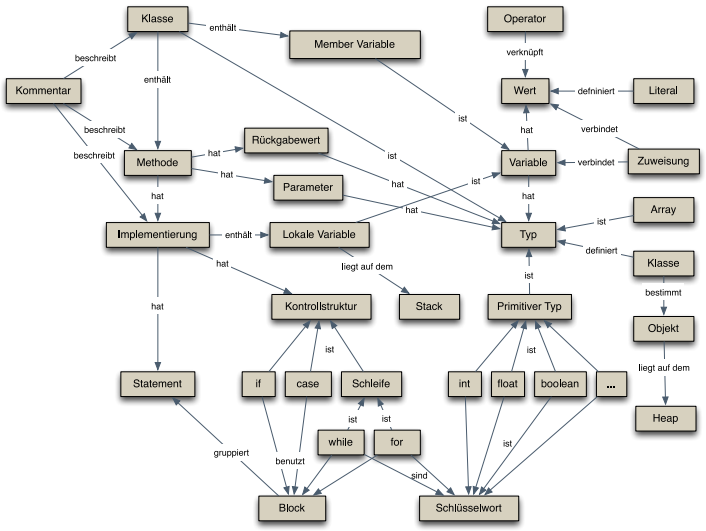
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Woran kann ich mich überhaupt noch aus PR1 erinnern?
- Welche Datentypen und Kontrollstrukturen hat Java?
- Wie funktioniert die Speicherverwaltung in Java?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 1. Java ist auch eine Sprache
- 2. Imperative Sprachkonzepte
- 4. Arrays und ihre Anwendungen

Kapitel 2

Typen und Operatoren

2.1 Java Schlüsselworte [9]

Java hat 50 **Schlüsselworte** (keywords)

Schlüsselworte

```
abstract   continue   for          new         switch
assert     default    if           package     synchronized
boolean    do         goto         private     this
break      double     implements   protected   throw
byte       else       import       public      throws
case       enum       instanceof   return      transient
catch      extends    int          short       try
char       final      interface    static      void
class      finally    long         strictfp    volatile
const      float      native       super       while
```

Java Schlüsselworte dürfen nicht für **Bezeichner** (identifier), d. h. Variablennamen, Klassennamen oder Methodennamen verwendet werden.

Bezeichner

Das Schlüsselwort `goto` ist zwar reserviert, wird aber in der Sprache gar nicht verwendet. Vermutlich hat man es zum Schlüsselwort gemacht, weil in der Sprache C `goto` ebenfalls ein Schlüsselwort ist (mit dem man unbedingte Sprünge machen kann) und man sich die Möglichkeit offen halten wollte, später Sprünge in Java einführen zu können. Da `goto` aber zum sogenannten Spaghetticode führt, ist es als positiv zu werten, dass Java solche Sprünge nicht unterstützt.

Im Rahmen dieser Vorlesung werden nahezu alle Schlüsselworte behandelt werden. Die einzige Ausnahme stellen `strictfp`, `native` und `assert` dar.

- `strictfp` kennzeichnet Methoden und Klassen, bei denen Gleitkommaberechnungen strikt nach den Regeln von IEEE745 ablaufen müssen.

  `strictfp`
- `native` kennzeichnet Methoden, die nicht in Java, sondern direkt nativ für die Plattform, z. B. in C++, entwickelt wurden.

  `native`
- `assert` erlaubt es, **Zusicherungen** (Assertions) direkt im Quelltext anzugeben.

  `assert`
  Zusicherungen

2.2 Kommentare [10]

```
// Dies ist ein Kommentar in einer Zeile

/* dies ist ein Kommentar
   über mehrere Zeilen
   (Blockkommentar) */

/** Hier kommt JavaDoc */

/** Hier kommt JavaDoc,
  * gerne auch Mehrzeilig */

/* Das geht nicht, da
   /* Mehrzeilige Kommentare nicht
      geschachtelt werden können
    */
*/
```

Java kennt drei Arten von Kommentaren:

- Einzeilige Kommentare, die vom `//` bis zum Ende der Zeile gehen
- Mehrzeilige Kommentare, die von `/*` und `*/` eingeschlossen werden müssen
- JavaDoc-Kommentare, die mit `/**` beginnen und mit `*/` enden

`//`

`/*`

Kommentare können nicht geschachtelt werden, d. h. es ist nicht möglich mehrere mehrzeilige Kommentare ineinander zu verschachteln. Eine Ausnahme sind einzeilige Kommentare, die von einem mehrzeiligen Kommentar umschlossen werden können, d. h. man kann schreiben:

```
/* Kommentar
   // Einzeiliger Kommentar
*/
```

2.3 Praxis-Tipp: Kommentare [11]

- Innerhalb von Methoden immer nur Zeilenkommentare verwenden
- So können Methoden einfach durch einen Blockkommentar auskommentiert werden

```
void methodeMitKommentaren() {
    // Erzeuge zwei Strings mit sinnvollem Inhalt
    String a = "Hallo";
    String b = "Welt";

    // Vertausche die Variablen
    swap(a, b);
```

```
    // Drucke Ergebnis aus
    System.out.println(a + " " + b);
}
```

Wenn Sie Eclipse verwenden, können Sie sehr komfortable Code aus- und wieder einkommentieren, indem Sie einfach CTRL-SHIFT-C drücken.

2.4 Primitive Datentypen [12]

Java hat eine Reihe von *eingebauten Datentypen*, die nicht durch Klassen repräsentiert werden. Sie werden **primitive Datentypen** (primitive types) genannt

primitive Datentypen

- Logisch – `boolean` — boolean
  - Zwei mögliche Werte: `true` und `false`
- Ganzzahl – `byte`, `short`, `int`, `long` — byte, short, int, long
  - 8, 16, 32, 64 Bit vorzeichenbehaftete Ganzzahlen
- Fließkomma – `float`, `double` — float, double
  - 32 bzw. 64 Bit Fließkommazahlen
- Zeichen – `char` — char
  - 16 Bit Unicode-Zeichen

Die primitiven Datentypen existieren in Java primär aus Performancegründen. Im Gegensatz zu anderen Sprachen (z. B. Ruby) hat man sich entschlossen, neben den klassenbasierten auch noch primitive Datentypen in Java zu verwenden. Da die primitiven Datentypen eine einfache Wertsemantik haben, kann man Operationen auf ihnen deutlich schneller ausführen als auf entsprechenden Klassentypen. Bei einem klassenbasierten Datentyp enthält die Variable nur eine Referenz auf das Objekt, das dann die Daten trägt. Jeder Zugriff auf die Variable im Objekt benötigt daher zwei Operationen: (1) Adresse des Objektes aus der Referenz lesen, (2) über die Adresse auf den Wert im Objekt zugreifen. Bei primitiven Datentypen enthält die Variable direkt den Wert, sodass man sich einen Schritt spart und der Zugriff damit schneller erfolgen kann.

Zu jedem primitiven Datentyp gibt es einen sogenannten Wrapper, d. h. einen korrespondierenden Typen, der auf einer Klasse basiert. Diese werden später in der Vorlesung noch behandelt.

Mit Java SE 5.0 wurde durch das Autoboxing ein Mechanismus eingeführt, bei dem der Compiler automatisch einen primitiven Typen in eine Wrapper umwandelt und umgekehrt, wenn an einer entsprechenden Stelle im Quelltext ein Objekt bzw. ein primitiver Typ benötigt wird. Dieses Feature werden wir später noch genauer beleuchten.

Bei `char` handelt es sich um ein einzelnes 16 Bit Unicode-Zeichen (in einer UCS2-Codierung), das nicht mit dem Klassentyp String verwechselt werden darf, der eine beliebig lange Kette

von Zeichen enthalten kann. Technisch gesehen ist ein `char` tatsächlich eine vorzeichenlose Zahl. Seit Java 5 wird der Unicode 4.0-Standard unterstützt, der 32 Bit pro Zeichen vorsieht. Da die Breite des Datentyps `char` aber auf 16 Bit beschränkt ist, hat man sich entschieden, Strings intern als UTF-16 abzubilden. Damit sind die allermeisten, gebräuchlichen Zeichen in 16 Bit abbildbar und für die wenigen Ausnahmen werden dann mehr als ein `char` zur Kodierung benötigt.

Bei den Fließkommazahlen handelt es sich um Datentypen, die für wissenschaftlich-technische Anwendungen gedacht sind, die aber gerade nicht in der Lage sind bestimmte Dezimalzahlen, z. B. bei der Arbeit mit Währungen, mit exakter Genauigkeit darzustellen. So ist es zum Beispiel nicht möglich, die Zahl 0.1 exakt mithilfe von Fließkommazahlen darzustellen, da es immer zu Rundungseffekten kommt. Wenn exakte Dezimalzahlen benötigt werden, muss man entweder mit `int` und `long` arbeiten und den Dezimalpunkt selbst verwalten oder auf die Klasse `BigDecimal` zurückgreifen.

2.5 Literale für Ganzzahlen [13]

- Vorzeichen
  - Positives Vorzeichen kann weggelassen werden: `476` oder `+467`
  - Negatives Vorzeichen ohne Leerzeichen: `-3521`
- Datentyp
  - Standardmäßig hat das Literal den Typ `int`
  - Für `long` muss ein `L` angehängt werden `7_000_000_000L`
- Prefix (ohne für Dezimalzahlen)
  - `0` für Oktalzahlen `0`
  - `0x` für Hexadezimalzahlen `0x`
  - `0b` für Binärzahlen `0b`

Der Prefix für Oktalzahlen ist eine häufige Falle für Programmierneulinge. In der Mathematik ist es völlig egal, ob man 42 oder 042 schreibt. Beides bezeichnet dieselbe Zahl. In Java ist das anders, da durch die führende 0 eine Oktalzahl angezeigt wird. 042 entspricht der Dezimalzahl 34.

Seit Java 8 darf man an beliebigen Stellen einen Unterstrich `_` zur Trennung der Ziffern einfügen.

2.6 Strings [14]

Für **String-Literale** sind die doppelten Anführungszeichen `"` reserviert, einfache Anführungszeichen `'` werden für Zeichen-Literale vom Typ `char` verwendet. Damit folgt Java der Tradition von C und weicht von anderen Sprachen wie JavaScript, Python, PHP und Ruby ab, bei denen Strings sowohl mit `'` als auch mit `"` angegeben werden können. String-Literale

`String` ist ein Referenzdatentyp in Java

- Literale mit "
- Verkettung mit +
- Mehrzeilige Literale mit """ (führende Leerzeichen werden entfernt)

"
+

```
String s1 = "Ein Literal";
String s2 = "Ein verkettetes\n"
          + "Literal";
String s3 = """
    Ein mehrzeiliger
    String""";
```

Strings sind *unveränderlich* (immutable). Dies bedeutet, dass Strings nach ihrer Erzeugung nicht mehr verändert werden können. Jede Änderung erzeugt ein neues `String`-Object.

Strings

```
String s1 = "Hallo";
String s2 = s1 + " Welt";
System.out.println(s2 == s1); // -> false
```

Das Beispiel zeigt, dass durch die Verkettung ein neuer String entsteht. Dies kann man sich syntaktisch noch gut erklären. Aber spätestens beim folgenden Beispiel erkennt man die Unveränderlichkeit deutlich.

```
String s = "Hallo";
s.replace('a', '4');
System.out.println(s); // -> "Hallo"
```

Die Ersetzung geht ins Leere, weil der Rückgabewert der `replace`-Methode (der neu erzeugte String) nicht „abgeholt“ wird.

2.7 StringBuilder und StringBuffer [16]

Dadurch, dass Strings unveränderlich sind, sind viele Operationen ineffizient. Baut man z. B. einen String schrittweise mit + zusammen, werden sehr viele unnötige Objekte erzeugt, die sofort wieder verworfen werden. Deswegen gibt es in Java zwei andere Klassen, die **veränderbare Strings** zur Verfügung stellen: `StringBuilder` und `StringBuffer`.

veränderbare Strings
StringBuilder
StringBuffer

`StringBuilder` und `StringBuffer`

- stellen veränderliche Strings zur Verfügung
- sind effizienter als die Klasse `String`
- sind, wie auch `String`, von `CharSequence` abgeleitet

```
StringBuilder sb = new StringBuilder();

sb.append("Hallo").append(' ').append("Welt");
System.out.println(sb.toString()); // -> Hallo Welt
```

Der Unterschied zwischen `StringBuilder` und `StringBuffer` liegt darin, dass `StringBuffer` *threadsicher* ist und `StringBuilder` nicht. Wenn es zu keinem konkurrierenden Zugriff (siehe unten) durch mehrere Threads kommt, ist die Klasse `StringBuilder` vorzuziehen, weil sie effizienter Arbeitet.

`equals` von `StringBuilder` und `StringBuffer` vergleicht nicht den Inhalt.

```
var sb1 = new StringBuilder("Hallo");
var sb2 = new StringBuilder("Hallo");
System.out.println(sb1.equals(sb2)); // -> false
```

2.8 Arrays sind Objekte [17]

- Deklaration einer Referenzvariablen auf ein Array
  ```
  char[] charArray;
  String[] stringArray;
  ```
- Arrays werden mit `new` angelegt und haben eine fixe Größe
  ```
  charArray = new char[10];
  stringArray = new String[10];
  ```
- Der Index eines Arrays beginnt bei 0
- Arrays können nicht mehr in der Größe verändert werden
- Array-Literale
  ```
  int[] intArray = { 1, 5, 7, 9 ,11, 22, 0 };
  String[] s = { "Alfred", "Franz", "Peter"};
  ```

Arrays sind Objekte und Array-Variablen sind ganz normale Referenzen auf ein Objekt. Daher wird durch das Deklarieren einer Array-Referenz das eigentliche Array nicht angelegt, sondern es wird nur eine Referenzvariable erzeugt. Damit verhalten sich Java-Arrays anders als Arrays z. B. in C wo bereits bei der Deklaration der Array-Variablen der Speicher für das Array beschafft wird. Die Array-Referenz hat den Wert `null` bzw. ist undefiniert, solange noch kein Array erzeugt und ihr zugewiesen wurde.

Neue Arrays werden wie Objekte mit `new` in erzeugt. Hierbei muss die Größe des Arrays angegeben werden, die dann nicht mehr verändert werden kann. Der einzige Weg ein Array zu

vergrößern ist es, ein neues, größeres Array anzulegen und den Inhalt umzukopieren. Hierzu dient die für diesen Zweck hochoptimierte `System.arraycopy()` Methode.

Die Elemente eines Arrays von primitiven Typen können direkt Daten aufnehmen. Sie sind nach der Erzeugung mit dem Standard-Wert des jeweiligen Typs initialisiert. Bei Arrays von Objekttypen sind die Elemente `null` und die entsprechenden Objekte müssen erst noch angelegt werden.

Da mehrdimensionale Arrays in Wirklichkeit Arrays von Arrays sind, kann man nicht rechteckige mehrdimensionale Arrays konstruieren.

```
int twoDim [][] = new int [4][];
twoDim[0] = new int[2];
twoDim[1] = new int[4];
twoDim[2] = new int[6];
twoDim[3] = new int[8];
```

2.9 Hilfsklasse Arrays [18]

Arrays sind zwar normale Objekte, sie haben aber keine Methoden, außer den von `Object` geerbten und dem read-only Attribut `length`. Diese Entscheidung lässt sich nachträglich auch nicht mehr ändern.

Es gibt allerdings die Klasse `Arrays` in `java.util`, welche eine ganze Reihe von hilfreichen Methoden für Arrays zur Verfügung stellt. Arrays

Hilfsmethoden für Arrays finden sich in der Klasse `java.util.Arrays`

- `sort`: Sortieren von Arrays
- `equals`: Vergleichen von Arrays
- `deepEquals`: Vergleichen von Arrays (deep)
- `fill`: Füllen eines Array mit Daten
- `copyOf`: Kopieren von Teilen eines Arrays
- `binarySearch`: Binäres suchen auf einem sortierten Array

Tatsächlich kann man zwei Arrays nur mit der `equals`-Methode von `Arrays` vergleichen und nicht mit der `equals`-Methode der Arrays selbst.

```
int[] a = { 1, 2, 3 };
int[] b = { 1, 2, 3 };
System.out.println(a.equals(b)); // -> false
System.out.println(Arrays.equals(a, b)); // -> true
```

Die `deepEquals`-Methode vergleicht auch Arrays, die wieder Arrays enthalten, indem Sie die inneren Arrays ebenfalls miteinander vergleicht.

2.10 Deklaration und Zuweisung [19]

```
// Deklaration
char c;
String str;
int x, y;

// Deklaration mit Initialisierung
float z = 1.712f;
double w = 3.1415;
boolean wahrheit = true;
String str1 = "Hi";
var v = 99;

// Zuweisung
c = 'Z';
str = "Hallo Welt";
x = 60;
y = 400;
```

Man kann mehrere Variablen in einer Zeile deklarieren, indem man die einzelnen Variablen durch Kommata trennt. Es ist auch möglich, eine Variable in der **Deklaration** direkt mit einem Wert zu versehen, d. h. Deklaration und **Zuweisung** zu kombinieren.

Deklaration
Zuweisung

Es ist sogar möglich, die Deklaration von normalen und Array-Variablen in einer Zeile zu mischen. Dies ist aber sehr unübersichtlich und fehleranfällig und sollte daher auf keinen Fall gemacht werden. Der Java Coding-Standard verbietet daher auch syntaktisch vollkommen korrekte Konstrukte wie das folgende:

```
int x, y[], z[][];
```

Sie können sogar in eine solche Deklaration noch eine oder mehrere Zuweisungen einbauen und damit die Lesbarkeit komplett zerstören:

```
int i = 5, j[] = { 1,2,3 }, k[][] = new int[2][3];
```

Beachten Sie, dass String-Literale in doppelten Anführungszeichen stehen, wohingegen Zeichen-Literale in einfachen Anführungszeichen geschrieben werden. Während das Zeichen-Literal einen primitiven Typen definiert, nämlich `char`, erzeugt ein String-Literal ein echtes Java-Objekt vom Typ `String`.

Der Java-Compiler stellt sicher, dass einer deklarierten lokalen Variablen auf jeden Fall ein Wert zugewiesen wurde, bevor sie das erste Mal benutzt wird. D. h. der folgende Code kompiliert nicht:

```
int x;
x++;
```

Die Fehlermeldung ist in diesem Fall „The local variable x may not have been initialized!“

2.11 Arten von Operatoren [20]

Drei Arten von Operatoren

- **Unäre Operatoren** haben nur einen Operanden — Unäre Operatoren
  - Syntax: `op Operand`
  - Beispiel: Negation einer Zahl (`a = -b`)
- **Binäre Operatoren** verknüpfen zwei Operanden — Binäre Operatoren
  - Syntax: `Operand1 op Operand2`
  - Beispiel: Multiplikation zweier Zahlen (`a = b * c`)
- **Ternäre Operatoren** verknüpfen drei Operanden (selten) — Ternäre Operatoren
  - Syntax: `Operand1 op Operand2 op Operande3`
  - Beispiel: Bedingter Ausdruck (`x = a > b ? a : b`)

Man kann Operatoren danach klassifizieren, wie viele Operanden sie haben. Am häufigsten sind Operatoren mit zwei Operanden (*binäre Operatoren*). Deutlich seltener sind solche mit nur einem Operanden (*unäre Operatoren*). Mit drei Operanden (*ternärer Operator*) existiert in Java nur ein einziger Operator, der sogenannte *Fragezeichen-Operator*.

2.12 Operatoren [21]

Assoziativität	Operator
R to L	++ – + - ~ ! (data type)
L to R	* / %
L to R	+ -
L to R	« » »>
L to R	< > ⇐ >= instanceof
L to R	! ==
L to R	&
L to R	^
L to R	|
L to R	&&
L to R	||
R to L	?:
R to L	*= /= %= += -= «= »= »>= &= ^= |= =

Da häufig mehrere Operatoren aufeinandertreffen, muss man festlegen, welche Operatoren zuerst ausgewertet werden. Diese Reihenfolge wird durch die *Rangfolge* der Operatoren ein-

deutig bestimmt, d. h. für jeden Operator ist festgelegt, vor welchen anderen Operatoren er ausgewertet wird.

Wenn ein Ausdruck mehrere Operatoren mit identischer Rangfolge enthält, wird die Richtung der Ausführung durch die **Assoziativität** bestimmt.

Assoziativität

Zum Beispiel sind Multiplikations- und Modulo-Operator linksassoziativ. Daher sind die beiden folgenden Anweisungen austauschbar:

```
int result = 5 * 70 % 6; // ergibt 2
int result = (5 * 70) % 6; // ergibt 2
```

Nicht aber

```
int result = 5 * (70 % 6); // ergibt 20
```

Rechtsassoziativ sind zum Beispiel die Vorzeichenoperatoren (+, –) oder die Zuweisung. Hier wird der Ausdruck von rechts nach links ausgewertet.

```
int summe;
int result = summe = 12; // summe ist 12, result ist 12
```

Kapitel 3

Kontrollstrukturen

3.1 Kontrollstrukturen: if [23]

```
int count = getCount(); // Methode im Programm

if (count < 0) {
    System.out.println("Fehler: Zahl ist negativ.");
} else if (count > getMaxCount()) {
    System.out.println("Fehler: Zahl ist zu groß.");
} else {
    System.out.println("Es kommen " + count + " Leute");
}
```

Generell ist die Syntax des `if`-Statements:

```
if (boolescher Ausdruck)
    Statement oder Block;
else if (boolescher Ausdruck)
    Statement oder Block;
else
    Statement oder Block;
```

Es kann beliebig viele `else if`-Zweige geben, aber nur einen `else`-Zweig.

3.2 Praxis-Tipp: if immer mit Block [24]

- Von der Syntax her greift die if-Kontrollstruktur immer für das nächste Statement oder den nächsten Block
- Schreiben Sie immer einen Block, auch wenn nur ein Statement unter der if-Bedingung stehen soll

```
// Beim Einfügen eines Statements geht es schnell schief
```

```
if (lottogewinn) {
    freuen();
    jobKuendigen();
}

if (lottogewinn)
    freuen();
    jobKuendigen(); // Wird immer ausgeführt!!!
```

Die Verwendung von `if` ohne Block ist unter C-Programmierer:innen sehr verbreitet, ist aber dort ebenfalls als gefährlicher Stil anzusehen. Programme sollten immer möglichst defensiv geschrieben sein, um spätere Überraschungen und Probleme zu minimieren. Dazu gehört ein Coding-Stil, bei dem man sich nicht selbst ein Bein stellt.

3.3 Kontrollstrukturen: switch [25]

```
switch (mitarbeiterArt) {

    case MANAGER:
        addEinzelnesBuero();
        addFirmenwagen();
        addSekretaerin();
        break;

    case SENIOR_DEVELOPER:
        addEinzelnesBuero();
        addFirmenwagen();
        break;

    default:
        addGrossraumbuero();
}
```

Die Variable, die im `switch`-Statement verwendet wird (hier `mitarbeiterArt`) muss entweder zuweisungskompatibel mit dem Typ `int` sein (`byte`, `short`, `char`, `int`) oder es muss sich um eine Referenz auf eine Enumeration handeln (Enumerationen werden später eingeführt). Es ist nicht möglich im `switch`-Statement Variablen von anderen Typen zu verwenden, d. h. `double`, `float`, `boolean` etc. sind nicht verwendbar. Der Grund hierfür liegt in der internen Implementierung von `switch` in der Java Virtual Machine. Seit Java 7 kann man `String` in einem `switch`-Statement verwenden (siehe unten).

`switch`

In den `case`-Ästen brauchen Sie keinen expliziten Block zu schreiben, d. h. anders als beim `if` bezieht sich der `case`-Ast auf alle Statements bis zum nächsten `case` oder `break`.

`case`

Der `default`-Ast wird ausgeführt, wenn keine der `case`-Konstanten gepasst hat.

`default`

Bei Java fällt bei eine `switch`-Statement der Kontrollfluss durch alle `case`-Statements, die auf das `case` folgen, bei dem die Bedingung zutraf. Daher muss man immer explizit ein `break`; hinter die `case`-Statements schreiben, wenn man dieses Verhalten nicht möchte, was fast immer der Fall ist. Java hat dieses Verhalten von C geerbt, wo es ebenfalls als obskur gilt.

Die allgemeine Syntax des `switch`-Statements ist:

```
switch (expr1) {
    case constant2:
        statements;
        break;
    case constant3:
        statements;
        break;
    default:
        statements;
        break;
}
```

3.4 Kontrollstrukturen: switch in Java 7 [26]

```
switch (titel) {
    case "Manager":
        addEinzelnesBuero();
        addFirmenwagen();
        addSekretaerin();
        break;

    case "Senior":
        addEinzelnesBuero();
        addFirmenwagen();
        break;

    default:
        addGrossraumbuero();
        break;
}
```

Mit Java 7 sind zusätzlich Variablen vom Typ String in `switch`-Statements erlaubt. Die Beschränkung aller vorhergehenden Java-Versionen auf Variablen, die sich nach `int` auswerten lassen, ist damit gefallen. Weiterhin kann man aber kein `long` oder `float` oder `double` verwenden.

3.5 Kontrollstrukturen: switch ab Java 12 [27]

Man merkt `switch` seine Abstammung von der Programmiersprache C deutlich an. Insbesondere die Verwendung von `break` und das Fehlen von geschweiften Klammern für die Instruktionen, die ausgeführt werden, wirken nicht „natürlich“. Aus diesen Gründen hat man in Java 12 eine neue Form von `switch` eingeführt, die syntaktisch anders aussieht und mehr nach „Java schmeckt“.

```
switch (titel) {
    case "Manager" -> {
        addEinzelnesBuero();
        addFirmenwagen();
        addSekretaerin();
    }
    case "Senior" -> {
        addEinzelnesBuero();
        addFirmenwagen();
    }
    default -> addGrossraumbuero();
}
```

Wenn nur ein einzelnes Statement in dem Zweig vorkommt, kann man die Klammern auch weglassen, wie es hier im `default`-Zweig gezeigt ist.

Wenn man im `switch` nur einen Wert zuweisen möchte, kann man die Kontrollstruktur als Ausdruck verwenden.

```
// old-school switch (Java < 12)
int tage;
switch (monat) {
    case 2: tage = 28; break;
    case 4:
    case 6:
    case 9:
    case 11: tage = 30; break;
    default: tage = 31;
}

// Neue Form (Java >= 12)
int days = switch (monat) {
    case 2 -> 28;
    case 4, 6, 9, 11 -> 30;
    default -> 31;
};
```

Mit Java 14 wurde dann auch noch die Syntax des alten Switch mit : erweitert, sodass man die Konstanten durch Komma (,) getrennt schreiben kann:

```
// new old-school switch (Java >= 14)
int tage;
switch (monat) {
    case 2:
        tage = 28;
        break;
    case 4, 6, 9, 11:
        tage = 30;
        break;
    default:
        tage = 31;
}
```

Problematisch wird die Zuweisung eines Wertes in der neuen Form, wenn man einen Block verwenden möchte, um noch etwas zu berechnen. Hierzu führt Java das neue Schlüsselwort `yield` ein, mit dem man in der neuen `case`-Variante etwas zurückgeben kann.

Neues Schlüsselwort `yield` für `case` yield

```
// case mit yield
int days = switch (monat) {
    case 2 -> { int d = 27; d++; yield d; }
    case 4, 6, 9, 11 -> 30;
    default -> 31;
};
```

3.6 Kontrollstrukturen: for-Schleife [30]

```
// Kindersimulator
for (int i = 0; i < 100; i++) {
    System.out.println("Sind wir schon da?");
    System.out.println("Ist es noch weit?");
}

System.out.println("Jetzt sind wir angekommen!");
```

Allgemein hat die for-Schleife die Form:

```
for (Initialisierung; Boolescher Test; Veränderungs-Ausdruck)
    statement oder block;
```

Wie bei `if` kann man bei `for` den Block weglassen, wodurch sich die `for`-Schleife auf das nächste Statement bezieht. Hiervon ist aber, genauso wie bei `if`, dringend abzuraten, da es den Source-Code erheblich schlechter lesbar macht. for

Man kann in der `for`-Schleife einzelne Teile weglassen. Daher sind die folgenden Schleifen korrekter Java-Code:

```
int i = 0;
for (; i < 10; i++) {
    System.out.println("Variable außerhalb deklariert");
}
```

```
int j = 0;
for (; j < 10;) {
    j++;
    System.out.println("Variable ausserhalb deklariert");
    System.out.println("Inkrement innerhalb");
}
```

```
for (;;) {
    System.out.println("Endlosschleife");
}
```

Da sie aber verwirrend sind, sollte man solche Formen normalerweise nicht verwenden.

Es ist zusätzlich möglich, in der `for`-Schleife mehrere Initialisierungen und Veränderungen parallel durchzuführen. Hierzu dient das Komma (`,`) als Trenner. Aber dieses Konstrukt ist ebenfalls mit Vorsicht zu genießen:

```
for (i = 0, j = 0; j < 10; i++, j++) { }
```

3.7 Kontrollstrukturen: for-each Schleife [31]

```
int[] primzahlen = { 2, 3, 5, 7, 11, 13, 17, 19 };

int summe = 0;

for (int primzahl : primzahlen) {
    summe += primzahl;
}
```

Die hier dargestellte foreach Schleife ist semantisch vollständig identisch mit der klassischen Variante, die einen Index verwendet, um über das Array zu iterieren.

```
for (int i = 0; i < primzahlen.length; i++) {
    int primzahl = primzahlen[i];
    summe += primzahl;
}
```

Der Java-Compiler erzeugt sogar aus der **foreach-Schleife** genau die oben dargestellte klassische Variante.

foreach-Schleife

3.8 Kontrollstrukturen: while-Schleife [32]

```
int i = 0;

while (i < 10) {
    System.out.println("Sind wir schon da?");
    System.out.println("Ist es noch weit?");
    i++;
}

System.out.println("Jetzt sind wir angekommen!");
```

Die `while`-Schleife hat die allgemeine Form:

while

```
while (Boolescher Ausdruck)
    Statement oder Block;
```

Der Rumpf der `while`-Schleife wird nur betreten, wenn die Bedingung des booleschen Ausdrucks wahr ist. Wenn man eine Schleife wünscht, die mindestens einmal durchlaufen wird, muss man die do/`while`-Schleife wählen. Die Variable auf die getestet wird, sollte richtig initialisiert sein und man sollte nicht vergessen sie im Schleifenrumpf zu verändern, da man andernfalls in einer Endlosschleife landet.

Wenn die Anzahl der Schleifendurchläufe schon im Voraus feststeht, sollten Sie die `for`-Schleife verwenden. Das hier gewählte Beispiel ist also nicht optimal, da man es besser mit `for` implementiert hätte. `while` ist gut geeignet für Schleifen, bei denen während des Durchlaufens erst festgestellt werden kann, wann der Abbruch erfolgen soll, z. B. bei linearen Suchen etc.

3.9 Kontrollstrukturen: do/while-Schleife [33]

```
int i = 0;
```

```
do {
    System.out.println("Sind wir schon da?");
    System.out.println("Ist es noch weit?");
    i++;
} while (i < 10);


System.out.println("Jetzt sind wir angekommen!");
```

Die **do/while**-Schleife hat die Form: do/while

```
do {
  Statement oder Block;
} while (Boolescher Ausdruck);
```

Der Rumpf der `do/while`-Schleife wird immer mindestens einmal betreten, da der Test erst am Ende erfolgt – es handelt sich also um eine fußgesteuerte Schleife. Wenn man eine Schleife wünscht, die die Bedingung vorher testet, muss man die `while`-Schleife verwenden. Die Variable auf die getestet wird sollte richtig initialisiert sein und man sollte nicht vergessen sie im Schleifenrumpf zu verändern, da man andernfalls in einer Endlosschleife landet.

Wenn die Anzahl der Schleifendurchläufe schon im Voraus feststeht, sollten Sie die `for`-Schleife verwenden.

3.10 break, continue und Labels [34]

Zusätzlich Schlüsselworte und Konstrukte

- `break` - Springt aus Schleifen und `switch`-Statements break
- `continue` - Springt an das Ende der Schleife und führt den Schleifen-Test erneut aus continue
- Zusätzlich kann man noch **Marken** (labels) im Code anlegen. Gibt man die Marke bei `break` oder `continue` an, wird nicht die aktuelle Schleife verlassen bzw. erneut ausgeführt, sondern die mit dem Label markierte Schleife Marken
- Marken können einen beliebigen Namen, gefolgt von einem Doppelpunkt haben, z. B. `aussen:`

3.11 Deklaration einer Klasse [35]

```
class Ding {

    // Attribut, Instanzvariable, Membervariable
    int x;

    // Methode
    int getX() {
```

```
        return x;
    }

    // Methode
    void setX(int newX) {
        x = newX;
    }
}
```

In Java werden Klassen deklariert, indem man das Schlüsselwort `class`, gefolgt vom Namen der Klasse schreibt. Im Rumpf der Klassendeklaration werden dann die Attribute und Methoden-Deklarationen geschrieben. Formal ist die Grammatik für die Deklaration einer Klasse:

class

```
<class_declaration> ::=
  <modifier> class <name> {
    <attribute_declaration>*
    <constructor_declaration>*
    <method_declaration>*
  }
```

Die Attribute, innerhalb einer Klasse werden definiert gemäß:

```
<attribute_declaration> ::=
  <modifier>  <type> <name> [=  <default_value>];

<type> ::= byte | short | int | long | char
           float | double | boolean |  <class>
```

Analog dazu werden Methoden geschrieben mit:

```
<method_declaration> ::=
  <modifier> <return_type> <name> (<parameter>*) {
    <statement>*
  }

<parameter> ::= <parameter_type> <parameter_name>,
```

Kapitel 4

Speicherverwaltung

4.1 Begriffe [37]

- **Hauptspeicher** (main memory): Speicherort für Informationen in einem Computer — Hauptspeicher
- Anderer Name *RAM* (random access memory)
- Information wird in Form von Bytes gespeichert (1 **Byte** = 8 Bit) — Byte
  00000000–11111111 bzw. 0x00–0xFF bzw. 0–255
- Byte ist die kleinste Einheit, die im Hauptspeicher ansprechbar ist
- Zwei gängige Technologien für RAM sind SRAM und DRAM

Der Arbeitsspeicher oder Hauptspeicher (engl.: main memory, main store, primary memory) [… ist die] Bezeichnung für den Speicher, der die gerade auszuführenden Programme oder Programmteile und die [dabei] benötigten Daten enthält. Der Hauptspeicher ist eine Komponente der Zentraleinheit. Da der Prozessor unmittelbar auf den Hauptspeicher zugreift, beeinflussen dessen Leistungsfähigkeit und Größe in wesentlichem Maße die Leistungsfähigkeit der gesamten Rechenanlage.

Quelle: Duden Informatik

4.2 Einheiten (alte Form) [38]

Missbräuchliche Verwendung der SI-Prefixe für Zweierpotenzen

- *Kilobyte* (kB) = 2^{10} Byte = 1024 Byte
- *Megabyte* (MB) = 2^{20} Byte = 1024 kB
- *Gigabyte* (GB) = 2^{30} Byte = 1024 MB
- *Terabyte* (TB) = 2^{40} Byte = 1024 GB
- *Petabyte* (PB) = 2^{50} Byte = 1024 TB
- *Exabyte* (EB) = 2^{60} Byte = 1024 PB

Die Prefixe des SI-Systems sind für Zehnerpotenzen reserviert. Da es aber bis zur Mitte der 1990er Jahre keine Prefixe für Zweierpotenzen gab, wurden häufig die SI-Prefixe „missbraucht“.

Dies hat leider zur Folge, dass nicht klar ist, was mit MB gemeint ist. Gerade die Hersteller von Festplatten interpretieren MB als 10^6 Byte, da sie dann die Kapazität höher angeben können.

4.3 Einheiten (neue Form) [39]

Einheiten nach EC 80000-13:2008 für dezimale Einheiten

- **Kilobyte** (kB) = 10^3 Byte = 1000 Byte — Kilobyte
- **Megabyte** (MB) = 10^6 Byte = 1000 kB — Megabyte
- **Gigabyte** (GB) = 10^9 Byte = 1000 MB — Gigabyte
- **Terabyte** (TB) = 10^{12} Byte = 1000 GB — Terabyte
- **Petabyte** (PB) = 10^{15} Byte = 1000 TB — Petabyte
- **Exabyte** (EB) = 10^{18} Byte = 1000 PB — Exabyte

Einheiten nach EC 80000-13:2008 für Zweierpotenzen

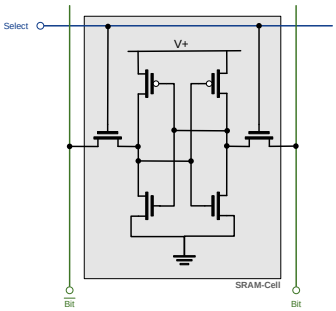
- **Kibibyte** (KiB) = 2^{10} Byte = 1024 Byte — Kibibyte
- **Mebibyte** (MiB) = 2^{20} Byte = 1024 KiB — Mebibyte
- **Gibibyte** (GiB) = 2^{30} Byte = 1024 MiB — Gibibyte
- **Tebibyte** (TiB) = 2^{40} Byte = 1024 GiB — Tebibyte
- **Pebibyte** (PiB) = 2^{50} Byte = 1024 TiB — Pebibyte
- **Exbibyte** (EiB) = 2^{60} Byte = 1024 PiB — Exbibyte

Die neuen Einheiten nach EC 80000-13:2008 unterscheiden sauber zwischen den dezimalen Prefixen und denen für Zweierpotenzen. Die neuen Einheiten haben sich aber nicht flächendeckend durchgesetzt, sodass immer noch nicht klar ist, welche Einheit jemand meint, wenn er von Megabyte oder Gigabyte spricht.

4.4 SRAM [41]

SRAM (static random-access memory) — SRAM

- Flipflop speichert ein einzelnes Bit
- sehr schnell aber teuer
- braucht keinen Refresh



Schaltbild einer SRAM-Zelle

Static random-access memory (deutsch: statisches RAM, Abkürzung: SRAM) bezeichnet einen elektronischen Speicherbaustein. Sein Inhalt ist flüchtig, das heißt, die gespeicherte Information geht bei Abschaltung der Betriebsspannung verloren. Im Gegensatz zu dynamischem Speicher (DRAM), welcher zur Vermeidung von Datenverlust ein periodisches Auffrischen (engl. refresh) benötigt, kann der Dateninhalt im statischen RAM bei Anliegen der Betriebsspannung beliebig lange gespeichert werden, wovon sich die Bezeichnung dieses Speichertyps ableitet.

Die Informationen werden durch Zustandsänderung von einer bistabilen Kippstufe in Form eines **Flipflops** pro Bit gespeichert. Das erlaubt es zwar, die Speicherzelle schnell auszulesen, aber im Vergleich zu dynamischen Speicherzellen ist die Speicherzelle verhältnismäßig groß. Im statischen Betrieb (Halten der Information) ist der Leistungsbedarf einer Zelle sehr klein.

Flipflops

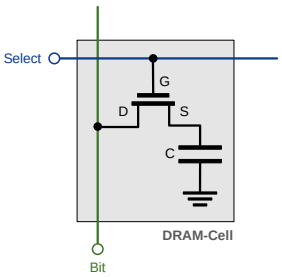
Quelle: Wikipedia

4.5 DRAM [42]

DRAM (dynamic random-access memory)

DRAM

- Kondensator speichert ein einzelnes Bit
- langsamer als SRAM aber deutlich billiger (höhere Packungsdichte)
- Speicher muss regelmäßig aufgefrischt werden, da Kondensator Ladung verliert



Schaltbild einer DRAM-Zelle

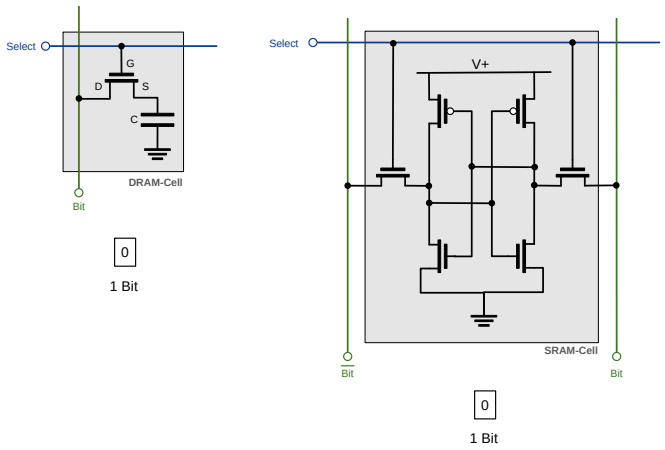
Dynamic Random Access Memory (*DRAM*) bezeichnet eine Technologie für einen elektronischen Speicherbaustein mit wahlfreiem Zugriff (Random Access Memory, RAM), der hauptsächlich in Computern eingesetzt wird. Das speichernde Element ist dabei ein Kondensator, der entweder geladen oder entladen ist. Über einen Schalttransistor wird er zugänglich und entweder ausgelesen oder mit neuem Inhalt beschrieben. Der Speicherinhalt ist flüchtig (volatil), das heißt die gespeicherte Information geht bei fehlender Betriebsspannung oder zu später Wiederauffrischung verloren.

Ein Kennzeichen des DRAM ist die Kombination aus einer *sehr hohen Datendichte* auf einer kleinen Chipfläche, verbunden mit sehr preiswerten Herstellungskosten. Diesem Vorteil des DRAM gegenüber dem SRAM steht der Nachteil gegenüber, dass sich die im Kondensator gespeicherte Ladung und damit die gespeicherte Information aufgrund von Leckströmen mit der Zeit verflüchtigt, wenn sie nicht periodisch wieder aufgefrischt wird. Dies ist normalerweise in Abständen von einigen Millisekunden erforderlich. Das *Auffrischen* des DRAMs verbraucht außerdem im Ruhezustand eine gewisse Menge von Energie. Deshalb bevorzugt man in Anwendungen, bei denen es auf geringen Ruhestrom ankommt, SRAM.

Quelle: Wikipedia

Über die *Select-Leitung* (blau) wird die Zelle ausgewählt, über die *Bit-Leitung* (grün) werden die Daten aus der Zelle gelesen bzw. in die Zelle geschrieben.

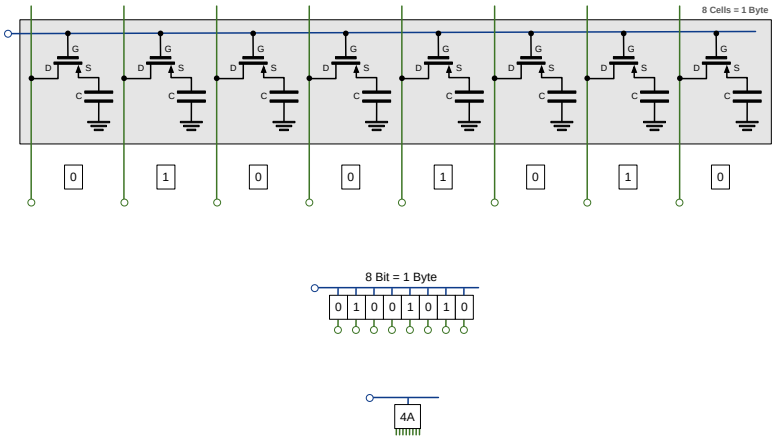
4.6 Vergleich DRAM / SRAM [45]



Vergleich einer DRAM- mit einer SRAM-Zelle

Man erkennt deutlich den Größenunterschied zwischen den beiden Arten von Speicherzellen. Während eine DRAM-Zelle nur einen einzigen Transistor benötigt, müssen in einer SRAM-Zelle sechs Transistoren verbaut werden. Wegen des drastischen Größenunterschiedes werden SRAM-Zellen nur dort verwendet, wo eine sehr hohe Performance nötigt ist, also vor allem für die Register des Prozessors und die Caches im Prozessor selbst.

4.7 DRAM-für ein Byte [46]



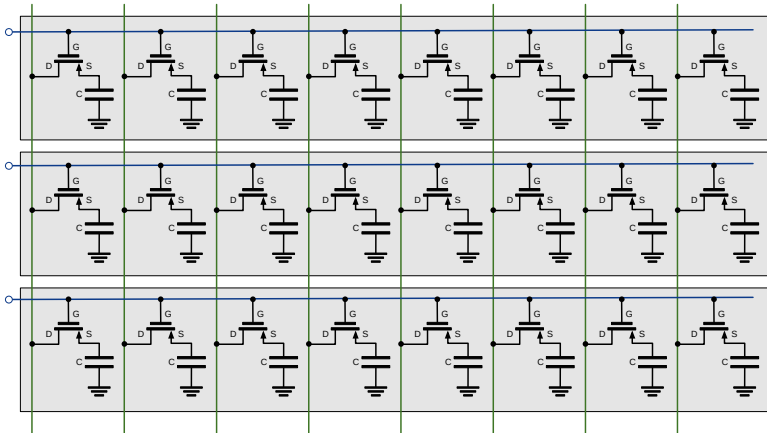
DRAM-Zeile

Durch das Zusammenschalten von acht DRAM-Zellen kann man acht Bit und damit ein Byte speichern. In allen modernen Computern ist der Hauptspeicher nicht Bit-, sondern Byte-

weise organisiert, d. h. es werden immer acht Bit gleichzeitig angesprochen. Eine Lese- oder Schreiboperation im Hauptspeicher adressiert immer mindestens ein Byte.

Das die Zellen für ein Byte immer gemeinsam adressiert werden erkennt man daran, dass sie sich eine Select-Leitung (blau) teilen.

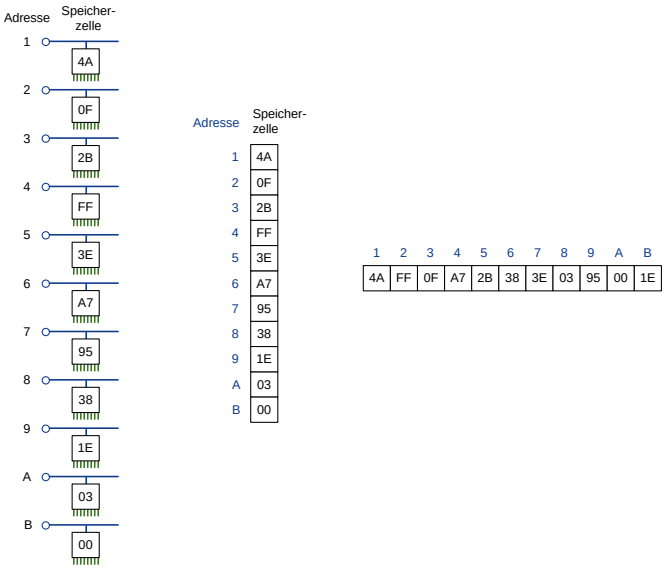
4.8 Speichergrid [47]



Speichergrid

Der Hauptspeicher eines Computers besteht aus sehr vielen Zellen, die jeweils ein Byte speichern.

4.9 Adresse [48]

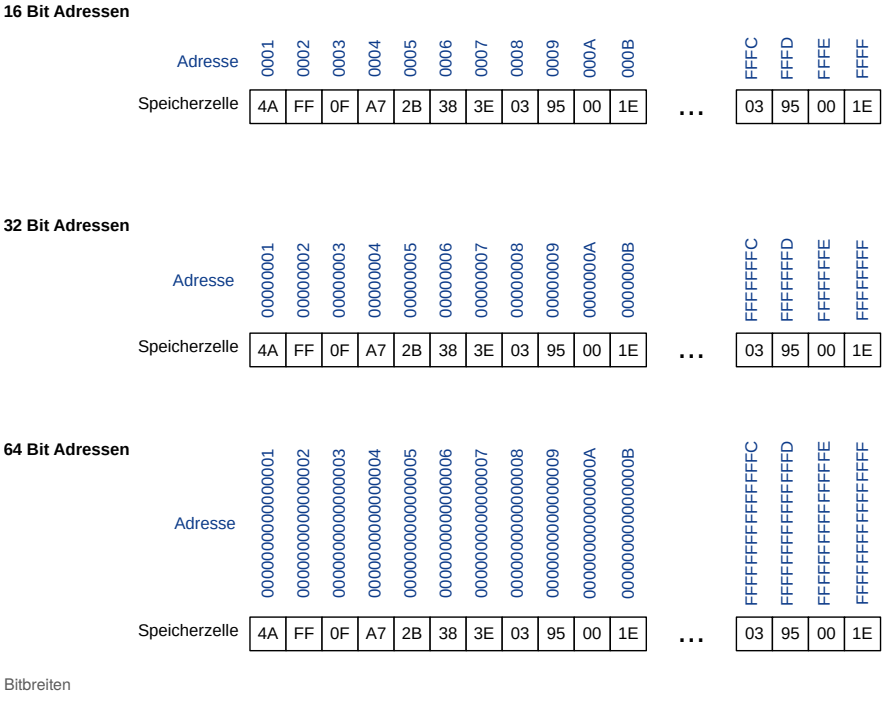


Adresse

Um die einzelnen Bytes im Hauptspeicher unterscheiden zu können, muss man ihnen eine eindeutige **Adresse** (address) geben. Hierzu werdend die Speicherstellen beginnend bei 1 durchnummeriert. Die Adresse ist somit selbst wieder nur eine Zahl, die eine Speicherstelle im Hauptspeicher eindeutig identifiziert. Wie alle Zahlen, mit denen ein Computer agiert, hat die Adresse eine feste Breite.

Adresse

4.10 Bitbreiten [49]



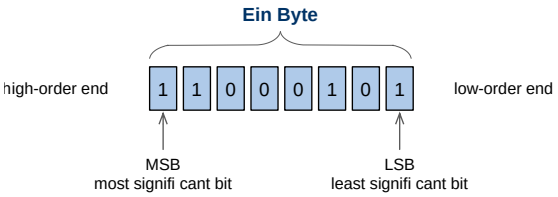
Bitbreiten

Eine wichtige Eigenschaft einer Computerhardware ist, wie lang die Adressen maximal werden können. Übliche Größen sind hier 16-Bit, 32-Bit und 64-Bit. Für PCs und Server ist heutzutage 64-Bit die übliche Größe, 32-Bit finden sich noch häufig bei Smartphones und 16-Bit trifft man bei Mikrocontrollern an.

Aus der Länge der Adresse ergibt sich, wie viel Speicher der Computer maximal verwalten kann.

- 16-Bit: 2^{16} Byte = 64 kByte
- 32-Bit: 2^{32} Byte = 4 GB
- 64-Bit: 2^{64} Byte = 16.777.216 TB

4.11 Speicherung eines Bytes [50]



Ein Byte

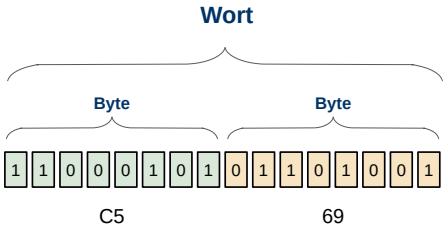
Wenn man ein Byte in den Speicherzellen des Hauptspeichers ablegt, stellt sich die Frage, wie man die einzelnen Bits anordnet. An einer Seite befindet sich das Bit mit der geringsten Wertigkeit, das **LSB** oder **least significant bit**, auf der anderen Seite das **MSB** bzw. **most significant bit**. Entsprechend bezeichnet man die beiden Seiten als **high-order end** und **low-order end**.

LSB
least significant bit
MSB
most significant bit
high-order end
low-order end

4.12 Größere Einheiten [51]

- **Wort** (word): 2 Byte, 16 Bit
- **Doppelwort** (double word, dword): 4 Byte, 32 Bit
- **Langwort** (long word): 8 Byte, 64 Bit

Wort
Doppelwort
Langwort



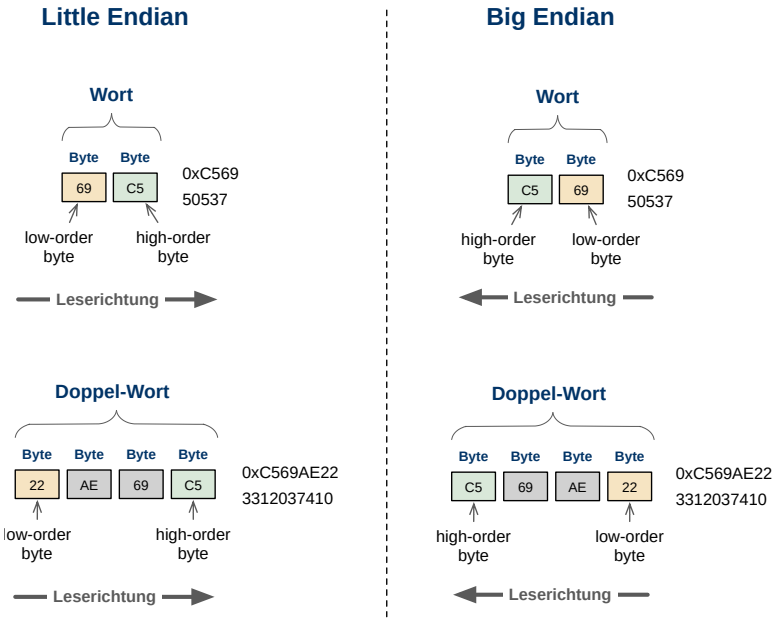
Word = 2 Byte

Viele Datentypen (siehe unten) benötigen mehr Platz, als ihn ein einziges Byte bietet. Daher haben sich für bestimmte Breiten (2, 4 und 8 Byte) eigene Begriffe etabliert.

4.13 Endianess [52]

- Für Worte und längere Datentypen stellt sich die Frage, wie man sie im Speicher ablegt
  - **Big Endian**: Das höchstwertige Byte wird zuerst geschrieben (an niedriger Adresse)
  - **Little Endian**: Das niederwertige Byte wird zuerst geschrieben (an niedriger Adresse)

Big Endian
Little Endian



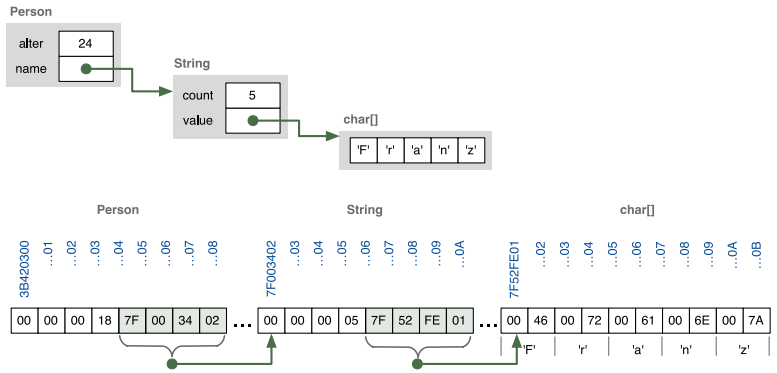
Big- vs. Little-Endian

Wenn Einheiten gespeichert werden, die größer als ein Byte sind (z. B. ein Wort), stellt sich sofort die Frage, an welcher Adresse die Bytes gespeichert werden, aus denen die Einheit besteht. Analog zu den Bits gibt es auch hier ein **low-order byte** und ein **high-order byte**. Leider gibt es bei den Computerherstellern keinen einheitlichen Standard, sodass es sowohl Computersysteme gibt, die das high-order byte an der niedrigeren Adresse speichern (Big Endian z. B. bei IBM PowerPC) und solche, die das high-order byte an der höheren Adresse speichern (Litle Endian z. B. bei Intel).

low-order byte
high-order byte

4.14 Speicherung von Objekten [54]

```
public class Person {
    int alter = 24;
    String name = "Franz";
}
```



4.15 Arten von Speicher [55]

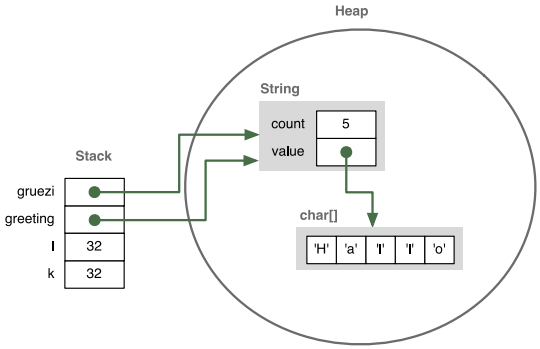
- **Stack** Stack
  - Kellerspeicher
  - Nimmt nur lokale Variablen auf
- **Heap** Heap
  - Wird vom Garbage Collector aufgeräumt
  - Nimmt nur Objekte auf

4.16 Objekt vs. Referenz [56]

Ein Objekt ist eine Instanz einer Klasse, die Daten und Methoden enthält. Eine Referenz ist eine Variable, die auf ein bestimmtes Objekt verweist. Ein Objekt selbst kann in den Arbeitsspeicher (Heap) geladen werden, während eine Referenz lediglich einen Verweis auf das Objekt enthält. Das bedeutet, dass mehrere Referenzen auf das gleiche Objekt verweisen können.

- **Objekt** (object) Objekt
  - liegt immer auf dem Heap
  - trägt Daten (primitive Datentypen oder Referenzen)
- **Referenz** (reference) Referenz
  - ist kein Objekt
  - zeigt auf ein Objekt
  - kann auf dem Heap liegen (als Variable in einem Objekt)
  - kann auf dem Stack liegen (als lokale Variable)
  - kann zu verschiedenen Zeiten auf verschiedene Objekte zeigen

```
int k = 32;
int l = k;
String greeting = new String("Hallo");
String gruezi = greeting;
```


Heap
String
count
5
value
Stack
gruezi
greeting
I
32
k
32
char[]
'H'
'a'
'l'
'l'
'o'

Kapitel 5

Quiz

5.1 Quiz: Wertebereich [59]

```
byte b = 127;
b++;

System.out.println(b);
```

```
-128
```

In Java sind alle primitiven Typen vorzeichenbehaftet, d. h. der Datentyp `byte` hat „nur“ einen Wertebereich von -128 bis 127. Darauf muss man beim Umgang mit primitiven Typen achten, da es keine vorzeichenlose Typen gibt und man daher z. B. den Wert 255 nicht mit einem `byte`, sondern nur mit einem `short` darstellen kann.

5.2 Quiz: Primitive Java Typen [61]

```
int i = 1;
double d;

d = i / 2;
System.out.println(d);

d = (double) i / 2;
System.out.println(d);

d = i / 2.0;
System.out.println(d);
```

```

```

```
0.0
0.5
0.5
```

Bei einer arithmetischen Operation wird in Java der Typ des Ergebnisses von den Operanden festgelegt und nicht vom Typ der Variable, an den das Ergebnis zugewiesen wird. Dividiert man einen Integer durch einen Integer, so ist das Ergebnis wieder ein Integer. Erst wenn mindestens einer der Operanden vom Typ double ist, wird im Beispiel das Ergebnis vom Typ `double` sein.

5.3 Quiz: Kontrollstrukturen [63]

```java
boolean b = false;

if (b = true) {
    System.out.println("Oops");
}
```



```
Oops
```



Im Gegensatz zu anderen Programmiersprachen (z. B. C oder C++) verlangt Java, dass der Ausdruck in der if-Bedingung immer zu einem `boolean` ausgewertet werden kann. D. h. man kann keinen Integer oder einen anderen Typ verwenden. Verwendet man allerdings eine `boolean` Variable, sollte man dringend davon Abstand nehmen, noch einen Vergleich mit == durchzuführen. Vergisst man hier nämlich ein Gleichheitszeichen, wird aus dem vermeintlichen Vergleich eine Zuweisung. Da jede Zuweisung selbst wieder einen Rückgabewert hat, nämlich den des rechten Operanden, kann man so aus Versehen ein syntaktisch korrektes if-Statement schreiben, das aber immer dem Wert des boolean-Literals folgt, mit dem man eigentlich vergleichen wollte.

```java
int x = 3;

switch (x) {
    case 1:
        System.out.print("1 ");
    case 2:
        System.out.print("2 ");
    case 3:
        System.out.print("3 ");
    case 4:
        System.out.print("4 ");
```



```
    default:
        System.out.print("Passt nicht");
}
```

```
3 4 Passt nicht
```

Bei Java fällt bei eine `switch`-Statement der Kontrollfluss durch alle case-Statements, die auf das `case` folgen, bei dem die Bedingung zutraf. Daher muss man immer explizit ein `break`; hinter die case-Statements schreiben, wenn man dieses Verhalten nicht möchte – was fast immer der Fall ist.

5.4 Quiz: Arrays [67]

```
int[] array = new int[5];

for (int i = 1; i < 6; i++) {
    array[i] = i;
}

System.out.println("Juhu!");
```

```
Exception in thread "main" ↩
    java.lang.ArrayIndexOutOfBoundsException: 5
     at pr2.Puzzles.main(Puzzles.java:70)
```

Array-Indices in Java beginnen bei 0 und laufen bis Länge des Arrays - 1. D. h. für ein Array der Länge `n` ist der gültige Bereich der Indices `0...n-1`. Im Beispiel wird aber das Array von 1 bis 5 durchlaufen, obwohl bei einem Array der Größe 5 nur die Indices von 0 bis 4 erlaubt wären.

Die Java-Laufzeitumgebung erzeugt bei einer Überschreitung der gültigen Array-Indices einen Laufzeit-Fehler (runtime exception) vom Typ `ArrayIndexOutOfBoundsException`.

5.5 Quiz: Logische Operatoren [69]

```
String s = null;

if ((s != null) & (s.length() > 0)) {
    System.out.println(s.charAt(0));
}
```

```
System.out.println(s);
```

```
Exception in thread "main" java.lang.NullPointerException
    at pr2.Puzzles.main(Puzzles.java:78)
```

Es gibt in Java zwei Arten von logischen Operatoren, die normalen | und & und die sogenannten Kurzschlussoperatoren || und &&. Bei den normalen Operatoren werden alle (Teil-)Ausdrücke ausgewertet und erst dann wird der logische Operator angewandt. Bei den Kurzschlussoperatoren wird die Auswertung streng von links nach rechts durchgeführt und sie wird abgebrochen, sobald der logische Wert des Gesamtausdrucks feststeht.

Die Verwendung der normalen Operatoren führt in dem Beispiel dazu, dass die null-Referenz dereferenziert wird, was zu einer NullPointerException führt. Hätte man im Beispiel satt & den Kurzschlussoperator && verwendet, wäre das Programm korrekt gelaufen.

Korrigierte Fassung

```
String s = null;

if ((s != null) && (s.length() > 0)) {
    System.out.println(s.charAt(0));
}

System.out.println(s);
```

5.6 Quiz: Parameterübergabe [71]

```
void swap(String a, String b) {
    String temp = a;
    a = b;
    b = temp;
}

void testSwap() {
    String a = "Hallo";
    String b = "Welt";
    swap(a, b);
    System.out.println(a + " " + b);
}
```

```
Hallo Welt
```

In Java werden Parameter an Methoden immer als Wert übergeben und nie als Referenz. D. h. die übergebene Variable wird kopiert und in der gerufenen Methode liegt eine Kopie dieser Variable vor. Diese Semantik gilt ebenfalls für Objektreferenzen; hier wird die Referenzvariable, die auf das Objekt zeigt, kopiert. Das Objekt selbst wird aber nicht kopiert. Daher kann die gerufene Methode zwar das Objekt manipulieren (was im gegebenen Beispiel wegen der Unveränderbarkeit von String allerdings nicht klappt) nicht aber den Wert der Referenz in der rufenden Methode ändern.

Aus diesem Grund geht die `swap()`-Methode im Beispiel komplett in die Leere und hat keinerlei Auswirkungen auf die rufende Methode `testSwap()`.

Index

Adresse, 26
Assoziativität, 11

Bezeichner, 2
Big Endian, 27
Binäre Operatoren, 10
Byte, 21

Deklaration, 9
do/while, 19
Doppelwort, 27
DRAM, 23

Exabyte, 22
Exbibyte, 22

Flipflops, 23
foreach-Schleife, 18

Gibibyte, 22
Gigabyte, 22

Hauptspeicher, 21
Heap, 29
high-order byte, 28
high-order end, 27

Kibibyte, 22
Kilobyte, 22

Langwort, 27
least significant bit, 27
Little Endian, 27
low-order byte, 28
low-order end, 27
LSB, 27

Marken, 19
Mebibyte, 22
Megabyte, 22
most significant bit, 27
MSB, 27

Objekt, 29

Pebibyte, 22
Petabyte, 22
primitive Datentypen, 4

Referenz, 29

Schlüsselworte, 2
SRAM, 22
Stack, 29
String-Literale, 5
Strings, 6

Tebibyte, 22
Terabyte, 22
Ternäre Operatoren, 10

Unäre Operatoren, 10

veränderbare Strings, 6

Wort, 27

Zusicherungen, 2
Zuweisung, 9

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Stil, JUnit und JavaDoc



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-29

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**
- 1.1 Fragen [5] . . . 1
- 1.2 Begriffe [6] . . . 1
- 1.3 Links zur Java-Insel [7] . . . 1

2 Programmierstil **2**
- 2.1 Programmierstil [9] . . . 2
- 2.2 Beispiel: Schlechter Stil [10] . . . 2
- 2.3 Motivation [11] . . . 2
- 2.4 Java Coding Standard [12] . . . 3
- 2.5 Beispiel: Guter Stil [13] . . . 3
- 2.6 Boolesche Rückgabewerte [15] . . . 4
- 2.7 Weitere Regeln [19] . . . 5

3 Unit-Tests mit JUnit 5 **7**
- 3.1 Unit Test [23] . . . 7
- 3.2 JUnit Annotations [24] . . . 7
- 3.3 JUnit Assertions [25] . . . 8
- 3.4 Beispiel: Unit die getestet werden soll [26] . . . 8
- 3.5 Beispiel: Unit-Test ohne JUnit [27] . . . 9
- 3.6 Beispiel: Unit-Test mit JUnit [28] . . . 9
- 3.7 JUnit und Pflichtübungen [29] . . . 10

4 Dokumentation mit JavaDoc **12**
- 4.1 Warum JavaDoc? [31] . . . 12
- 4.2 Beispiel: JavaDoc Ausgabe [32] . . . 13
- 4.3 Wie funktioniert JavaDoc? [33] . . . 13
- 4.4 Formatierung von JavaDoc [34] . . . 13
- 4.5 Beispiel: JavaDoc von String [35] . . . 14
- 4.6 JavaDoc und Pflichtübungen [37] . . . 15

5 Versionsverwaltung mit Git **16**
- 5.1 Das Problem [39] . . . 16
- 5.2 Die Lösung: Versionsverwaltung (VCS) [40] . . . 16
- 5.3 Zentrales VCS [41] . . . 17

5.4 Verteiltes VCS [42] 17
5.5 Wichtige Begriffe und Konzepte [43] 17
5.6 Zentrale Versionsverwaltungssysteme [45] 18
5.7 Verteilte Versionsverwaltungssysteme [46] 18
5.8 Git [47] 18
5.9 Beispiel: Git lokal verwenden [48] 19
5.10 Beispiel: Git verteilt verwenden [54] 21
5.11 Git und Eclipse [56] 22

Index **ii**

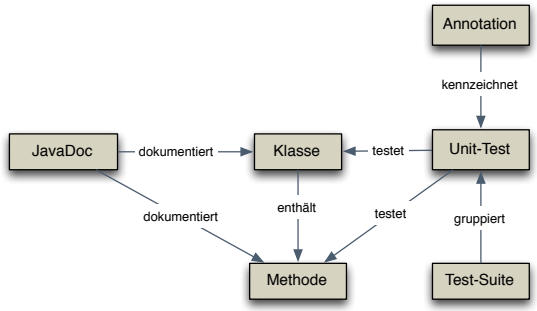
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie sichert man die Qualität von Software?
- Wie lasse ich Programme sich selbst testen?
- Wie programmiere ich wie ein Pro und nicht mehr wie ein Noob?
- Wie sorge ich dafür, dass andere meine Programme nutzen können?

1.2 Begriffe [6]



1.3 Links zur Java-Insel [7]

- 24.4 Dokumentationskommentare mit Javadoc
- 23. Testen mit JUnit

Kapitel 2

Programmierstil

2.1 Programmierstil [9]

> Talk is cheap, show me the code
>
> Linus Torvalds

> Don't comment bad code – rewrite it!
>
> Brian W. Kernighan

2.2 Beispiel: Schlechter Stil [10]

```
public class Div{static $1 $_;class $1{void _$(String $_){System.//
out.print($_);}$1(){_();}void _(){int _,$,_$,$$,__,ä=(1<<5),
Ã¥=100,ö=12,Ã¥ö=ä*ö;Ã¥-=1<<4;while(Ã¥ö>0){for($$=_$=_=__=$=(int)
Ã¥;_$>($$-(1<<2));_$(""+(char)(_$)),_$-=1<<1)for(_=$=__-9;_>(
$-6);_-=1<<1,_$(""+(char)(_$!=Ã¥?_:ä)));char S$=(char)(Ã¥+ö+1)
;_$("te"+S$+(char)(ö+(int)S$));Ã¥ö--;}}}Div(){$_=new $1();}
public static void main(String []$){Div Ã¥=new Div();}
```

2.3 Motivation [11]

Nicht der Compiler ist das Publikum, sondern Menschen, die den Source Code lesen müssen!

Probleme mit schlechtem **Programmierstil**

Programmierstil

- Teamarbeit wird durch inkonsistenten Stil erschwert
- Programme sind fehleranfälliger
- Änderungen am Programm werden erschwert
- Lesen des Source-Codes ist eine Qual
- Punktabzug bei den Pflichtübungen ;-)

Es gibt mehrere Gründe, warum es wichtig ist, einen guten, einheitlichen Programmierstil zu haben:

1. *Lesbarkeit*: Wenn ein(e) Programmierer:in einen konsistenten Stil verwendet, der leicht zu lesen und zu verstehen ist, können andere Entwickler:innen schneller erkennen, was der Code macht und wie er funktioniert. Dies führt zu einem schnelleren Verständnis und einer schnelleren Fehlerbehebung.
2. *Wartbarkeit*: Ein einheitlicher Programmierstil erleichtert die Wartung des Codes, da es einfacher ist, Fehler zu finden und Änderungen vorzunehmen.
3. *Zusammenarbeit*: Ein konsistenter Stil erleichtert die Zusammenarbeit und das Verständnis des Codes, da alle Entwickler:innen den Code auf die gleiche Weise lesen und interpretieren.
4. *Effizienz*: Wenn Entwickler:innen denselben Stil verwenden, können sie schneller arbeiten, da sie sich nicht mit unterschiedlichen Stilen auseinandersetzen müssen. Dies führt zu einer höheren Effizienz und einer schnelleren Entwicklung von Software.
5. *Professionalität*: Ein konsistenter Stil zeigt, dass die Entwickler:innen professionell sind und dass sie sich Gedanken darüber gemacht haben, wie der Code präsentiert wird.

2.4 Java Coding Standard [12]

Code grundsätzlich nach **Coding-Standard** formatieren! Coding-Standard

- *Keine TABs* verwenden, *Einrückung* pro Ebene *4 Leerzeichen*
- *Zeilenlänge* maximal *80 Zeichen*
- *Klassennamen* beginnen mit Großbuchstaben
- *Methodennamen* beginnen mit Kleinbuchstaben
- *Variablennamen* beginnen mit Kleinbuchstaben
- *KONSTANTEN* werden großgeschrieben
- Pro Zeile nur *eine Deklaration* bzw. *ein Statement*
- `CamelCase` für zusammengesetzte Namen

Sie finden den Coding-Standard für Java hier.

2.5 Beispiel: Guter Stil [13]

```java
package de.smits_net.pr2;

/**
 * Klasse dient als Beispiel für den Coding-Standard und die
 * Namenskonventionen bei Java-Programmen.
 *
 * @author Thomas Smits
 */
public class CodingStandard {
```

```
    /**
     * Konstante, die dem Rest der Welt etwas mitteilen soll.
     */
    public static final int KONSTANTE_MIT_TOLLEM_WERT = 3;

    private int erstesFeld;
    private double zweitesFeld;
```

```
    /**
     * Methode, die etwas tut - oder nicht ;-).
     *
     * @param parameter Eingabewert für die Methode
     * @return gibt immer 42 zurück
     */
    public int methodeDieWasTut(int parameter) {

        if (parameter > 3) {
            erstesFeld = 12;
        }
        else {
            zweitesFeld = 13;
        }

        return 42;
    }
}
```

2.6 Boolesche Rückgabewerte [15]

So schön Kontrollstrukturen sind, man braucht sie nicht immer. Ein klassischer Fehler ist, mit einem `if` einen booleschen Rückgabewert zu bestimmen. Solche Konstrukte kann man immer durch einen einfachen Ausdruck ersetzen.

Schlecht

```
if (bedingung) {
  return true;
}
else {
  return false;
}
```

Gut

```
return bedingung;
```

Schlecht

```
if (bedingung) {
  return false;
}
else {
  return true;
}
```

Gut

```
return !bedingung;
```

Schlecht

```
return bedingung ? true : false;
```

Gut

```
return bedingung;
```

Schlecht

```
return bedingung ? false : true;
```

Gut

```
return !bedingung;
```

2.7 Weitere Regeln [19]

Minimale Sichtbarkeit

- **Minimale Sichtbarkeit** von Methoden und Feldern (nur `public` wenn wirklich nötig)
- Verbergen Sie Implementierungsdetails (*Information Hiding*)
- *Dokumentieren* Sie alle öffentlichen Methoden mit JavaDoc

Don't Repeat Yourself

- Kopieren Sie keinen Code – **Don't Repeat Yourself** (DRY): Doppelter Code ist eine Hauptquelle von Fehlern
- Machen Sie *keine Ausgaben auf der Konsole* (Ausnahme, die Aufgabe verlangt es explizit)
- Lesen Sie *keine Daten von der Konsole* ein (Scanner ade)
- *Testen* Sie Ihre Klassen mit JUnit-Tests

- Programmieren Sie mit so *wenig Seiteneffekten* wie möglich
- Machen Sie Klassen (wenn möglich) *immutable*
- Denken Sie genau nach, was Sie in Attributen und was Sie in lokalen Variablen speichern

- *Fangen Sie* mögliche *Fehler* korrekt ab
- Denken Sie vor allem an die *Randfälle* und vertrauen Sie niemals den Eingabedaten
- Halten Sie *Daten und Methoden zusammen* (Abstrakte Datentypen)
- Vermeiden Sie komplexen und komplizierten Code
- Kommentieren Sie keinen Code aus: verwenden Sie ein Versionsverwaltungssystem, um alte Stände vorzuhalten
- Lassen Sie *keine* `// TODO`*-Kommentare* im Quellcode stehen
- Erzeugen Sie *keine leeren Catch-Blöcke*
- Benennen Sie Methoden nach dem Schema *Verb* oder *Verb + Nomen*, z. B. `copy`, `copyFile`, `removeEntry`
- Verwenden Sie Konstanten anstatt **magischer Werte**

magischer Werte

Einige der Regeln greifen vor, d. h. die dahinterliegende Theorie ist noch gar nicht eingeführt. Trotzdem werden sie hier bereits erwähnt, damit am Ende der Vorlesung alle Regeln an einer Stelle zusammengefasst sind.

Magische Werte sind Werte (z. B. `42`), die direkt im Code verwendet werden, ohne als Konstanten definiert zu werden. Diese Werte können schwer zu finden und zu ändern sein und können zu Fehlern führen, wenn sich ihre Bedeutung im Laufe der Zeit ändert. Benannte Konstanten (z. B. `ANTWORT_AUF_ALLES`) hingegen können leicht gefunden und geändert werden, da sie an einer zentralen Stelle definiert sind und an verschiedenen Stellen im Code verwendet werden können. Obwohl magische Werte möglicherweise weniger Code erfordern, ist es wichtig, benannte Konstanten zu verwenden, um Fehlerquellen zu minimieren.

Kapitel 3

Unit-Tests mit JUnit 5

3.1 Unit Test [23]

- Ein **Unit-Test** testet eine kleine selbständige Einheit (Unit) der Software, z. B. eine Klasse, ein API, ein Modul — Unit-Test
- Der Test versucht die Unit möglichst *isoliert* zu testen
- Unit Test werden häufig *vom Entwickler* selbst *geschrieben*
- Unit Tests sind ein wichtiger Teil der *Testgetriebenen Entwicklung*
- Für Java ist das gängigste Unit-Test-Framework **JUnit** — JUnit
  http://www.junit.org

Eine detaillierte Einführung in das Schreiben von Unit-Tests kann hier aus Zeitgründen nicht gegeben werden. In der Praxis gibt es eine ganze Reihe von Hilfsmitteln, um die zu testende Einheit zu isolieren und dann einzeln zu testen. Hierzu gehören Mocks, Stubs, Test Harness etc.

Die **Testgetriebene Entwicklung** (test-driven development) (**TDD**) ist eine Methode, die häufig bei der agilen Entwicklung von Computerprogrammen eingesetzt wird. Hier erstellt die Software-Entwickler:innen die Tests grundsätzlich *vor* den zu testenden Komponenten. — Testgetriebene Entwicklung, TDD

Die Menge der Tests, die zusammengehören, nennt man **Test-Suite**. — Test-Suite

3.2 JUnit Annotations [24]

- JUnit 5 arbeitet mit Annotations, sodass man keine Klassen ableiten oder Interfaces implementieren muss
- Man annotiert parameterlose Methoden und macht sie hierdurch zu Tests bzw. Hilfsmethoden für die Tests
- Die wichtigsten Annotations sind
  - `@Test` – Kennzeichnet eine Methode als Test — `@Test`
  - `@BeforeEach` – Methode wird vor den Tests ausgeführt (Test-Setup) — `@BeforeEach`
  - `@AfterEach` – Methode wird nach den Tests ausgeführt (Test-Teardown) — `@AfterEach`
  - `@Disabled` – Test wird (temporär) ignoriert und nicht ausgeführt — `@Disabled`

- `@DisplayName` – Name der Methode in der Anzeige

`@DisplayName`

Annotationen sind ein Java-Feature, das es erlaubt Klassen, Methoden und Attribute mit zusätzlichen Informationen zu versehen, die dann von anderen Klassen ausgewertet werden können. Eine Annotation verändert nicht das Verhalten eine Methode, liefert aber Meta-Informationen über diese. Man kann sich Annotationen grob als Kommentare vorstellen, die maschinenlesbar sind, d. h. die vom Programm selbst oder dem Compiler wieder ausgewertet werden können.

Annotationen

3.3 JUnit Assertions [25]

- Die Erwartungen des Tests drückt man durch Assertions aus
- Die Assertions liegen in der Klasse `org.junit.jupiter.api.Assertions`

Assertion	Bedeutung
`assertArrayEquals`	Elemente des Arrays sind gleich
`assertEquals`, `assertNotEquals`	Parameter sind gleich / ungleich
`assertSame`, `assertNotSame`	Objekte sind identisch / nicht identisch
`assertNull`, `assertNotNull`	Referenz ist null / nicht null
`assertTrue`, `assertFalse`	Bedingung ist wahr / falsch
`fail`	Schlägt auf jeden Fall fehl

Eine **Zusicherung** oder **Assertion** (lateinisch/englisch für Aussage, Behauptung) ist eine Aussage über den Zustand eines Computer-Programms. Mithilfe von Zusicherungen können logische Fehler im Programm erkannt und das Programm kontrolliert beendet werden. Des Weiteren können Assertions Informationen über den Grad der Testabdeckung während der Verifikation liefern. Quelle: Wikipedia

Zusicherung
Assertion

3.4 Beispiel: Unit die getestet werden soll [26]

```java
public class KomplexeZahl {

    public KomplexeZahl(double re, double im) { ... }

    public double getRe() { ... }
    public double getIm() { ... }

    public KomplexeZahl addiere(KomplexeZahl z) { ... }
    public KomplexeZahl subtrahiere(KomplexeZahl z) { ... }
    public KomplexeZahl multipliziere(KomplexeZahl z) { ... }
    public KomplexeZahl dividiere(KomplexeZahl z) { ... }

    public int hashCode() { ... }
    public boolean equals(Object obj) { ... }
```

```
    public String toString() { ... }
}
```

Die Klasse repräsentiert eine Komplexe Zahl und erlaubt es, mit diesen Zahlen einfache arithmetische Operationen durchzuführen. Die Implementierung ist hier weggelassen, da es für die Tests ausschließlich um die Schnittstelle, also die nach außen hin sichtbaren Methoden geht.

3.5 Beispiel: Unit-Test ohne JUnit [27]

```
public class KomplexTest {
    public void testAddition() {
        KomplexeZahl k1 = new KomplexeZahl(3, 2);
        KomplexeZahl k2 = new KomplexeZahl(5, 5);
        if (k1.addiere(k2).equals(new KomplexeZahl(8, 7))) {
            System.out.println("OK: Addition");
        }
        else {
            System.err.println("Fehler: Addition");
        }
    }

    public static void main(String[] args) {
        KomplexTest test = new KomplexTest();
        test.testAddition();
    }
}
```

Man kann Tests auch ohne entsprechendes Framework entwickeln, wie das obige Beispiel zeigt. Der Nachteil ist aber, dass man in diesem Fall selbst die Ausgaben kontrollieren muss und entscheiden, ob ein Fehler vorliegt oder nicht. Bei größeren Programmen oder häufigen Testläufen kann dies sehr mühsam werden.

3.6 Beispiel: Unit-Test mit JUnit [28]

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class KomplexeZahlTest {

    @Test
    void testAddition() {
        KomplexeZahl k1 = new KomplexeZahl(3, 2);
        KomplexeZahl k2 = new KomplexeZahl(5, 5);
```

```
        assertEquals(new KomplexeZahl(8, 7), k1.addiere(k2));
    }
}
```

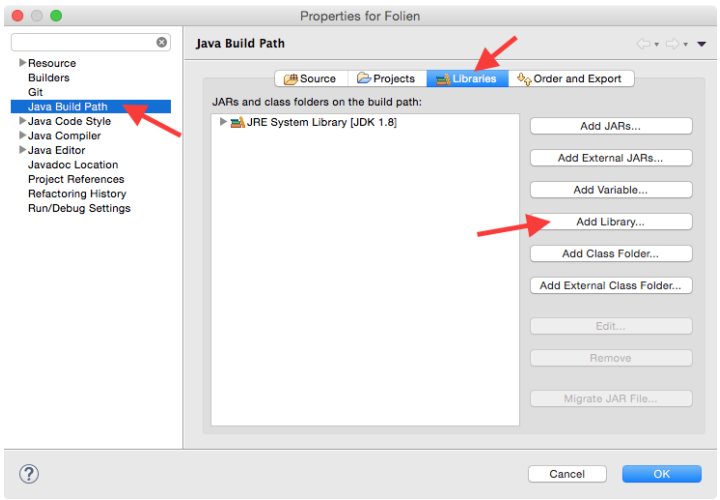
Der Test wird deutlich einfacher, wenn man das JUnit-Framework einsetzt. Zum einen kann man die Erwartungen an die korrekte Programmausführung klarer erkennen, da sie nicht mehr in einem `if` versteckt ist, sondern durch das `assertEquals(new KomplexeZahl(8, 7), k1.addiere(k2))` formuliert wird. Zum anderen wird der Test viel kürzer und kann jetzt automatisch ausgewertet werden. Die Entwickler:innen müssen nicht mehr auf jede einzelne Ausgabe schauen, es reicht sich das Gesamtergebnis anzusehen.

Bei JUnit 5 müssen die Testmethoden nicht mehr `public` sein, wie noch bei JUnit 4.

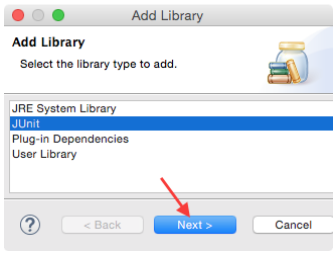
3.7 JUnit und Pflichtübungen [29]

- Es wird erwartet, dass Sie in den Pflichtübungen die Funktionalität Ihres Programms mit JUnit-Tests nachweisen
- Abgaben, bei denen die eigenen JUnit-Tests nicht funktionieren gelten als nicht bestanden
- Verwenden Sie bitte JUnit 5 und keine ältere Version

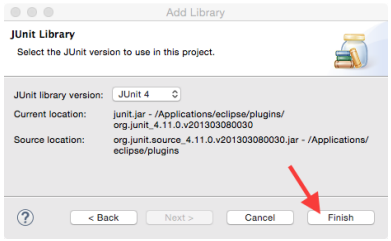
Sie können JUnit ganz einfach als Library Ihrem Eclipse-Projekt hinzufügen. Gehen Sie hierzu mit der rechten Maustaste auf das Projekt und wählen Sie „Properties“. Wählen Sie im Dialog dann „Java Build Path“, „Libraries“ und klicken Sie auf „Add Library“.



Wählen Sie im Folgenden Dialog „Junit“ aus und klicken Sie auf „Next“.



Nun wählen Sie „Junit 5“ und klicken auf „Finish“.



Kapitel 4

Dokumentation mit JavaDoc

4.1 Warum JavaDoc? [31]

Das Problem

- Java kennt keine Header-Dateien, die `.class`-Dateien sind ihre eigenen Header (selbstbeschreibend)
- Schnittstellen müssen aber dokumentiert werden, damit sie richtig verwendet werden können
- Wir wollen den Source-Code nicht weitergeben, dort steht aber die Dokumentation der Schnittstelle

Die Lösung

- Aus dem Source-Code eine HTML-Dokumentation generieren, die der Verwender der Klassen benutzen kann

In vielen Programmiersprachen (z. B. C++) gibt man für jede Klasse zwei Dateien weiter: zum einen die kompilierte Klasse und zum anderen eine sogenannte **Header-Datei**, die die Methoden beschreibt, die man von außen aufrufen kann. In der Header-Datei stehen alle Informationen dazu, wie man eine Methode verwendet.

Header-Datei

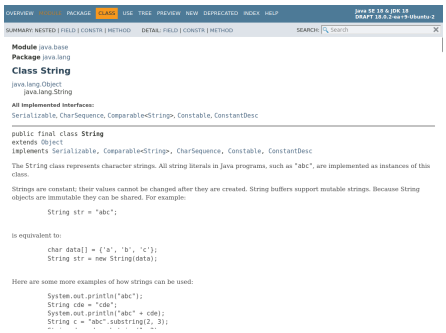
Java hat sich für eine andere Lösung entschieden: hier schreibt der Compiler alle Informationen, die man zur Nutzung einer Klasse braucht, direkt in die kompilierte Klasse hinein. Man benötigt also nur die `.class`-Datei, um mit der Klasse zu arbeiten und sie zu verwenden.

In der `.class`-Datei findet sich aber nur die technische Beschreibung der Schnittstelle aber keine Informationen zu den Namen der Parameter und keine Dokumentation der Methoden. Beim Kompilieren gehen also die Kommentare, der Entwickler:innen im Quelltext, verloren. Damit stellt sich die Frage, wie man die Dokumentation der Methoden zum Verwender der Klasse transportiert.

Die Lösung findet sich in der **JavaDoc**: aus speziellen Kommentaren im Quelltext erzeugt ein Werkzeug HTML-Seiten mit Dokumentation, die man dann zusammen mit den kompilierten Klassen weitergeben kann.

JavaDoc

4.2 Beispiel: JavaDoc Ausgabe [32]


Module java.base
Package java.lang
Class String
java.lang.Object
java.lang.String
All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>, Constable, ConstantDesc
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
The String class represents character strings. All string literals in java programs, such as "abc", are implemented as instances of this class.
Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:
String str = "abc";
is equivalent to:
char data[] = {'a', 'b', 'c'};
String str = new String(data);
Here are some more examples of how strings can be used:
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2, 3);

JavaDoc der Klasse String

4.3 Wie funktioniert JavaDoc? [33]

- Spezielle Kommentare vor Klassen und Methoden
  `/** JavaDoc Kommentar */`
- Tags, um Informationen zu kennzeichnen
  - `@author` - Author der Klasse — `@author`
  - `@version` - Version der Klasse — `@version`
  - `@throws` - Wann eine Exception geworfen wird — `@throws`
  - `@param` - Parameter eine Methode — `@param`
  - `@return` -Rückgabewert einer Methode — `@return`
  - `@see` - Querverweis — `@see`
- Programm (`javadoc`): liest Klassen, wertet Tags aus und erzeugt Dokumentation

Methoden sollten im Allgemeinen durch einen Satz mit Verb beschrieben werden, der die Aktionen beschreibt, die von der Methode ausgeführt werden, z. B. „Gibt die Anzahl der Elemente in der Liste zurück.“

Klassen, Interfaces und Variablen werden eher durch einen Satz mit Nomen beschrieben, der erklärt, was sie darstellen, z. B. „Eine Liste von Elementen.“

4.4 Formatierung von JavaDoc [34]

- Der erste Satz des JavaDoc-Kommentars ergibt die Zusammenfassung für die Übersicht
- Man kann HTML-Tags verwenden, um den Text zu strukturieren
  - `<p>...</p>`
  - `<i>...</i>`
  - `<pre>...</pre>`
  - `<code>...</code>`
- Seit Java 5 gibt es spezielle JavaDoc-Tags, die viele der HTML-Tags überflüssig machen

- {@code ... } {@code ... }
- {@literal ... } {@literal ... }

Als ersten Satz fasst das JavaDoc-Tool alle Zeichen am Anfang des Kommentars bis zum ersten Punkt auf, den ein Leerzeichen, Tab oder Zeilenende folgt auf. Dieser Teil wird dann in die Klassenübersicht übernommen und als Zusammenfassung der Methodenbeschreibung gewertet. Grundsätzlich sollten sich die Zusammenfassungen aller Methoden einer Klasse bzw. eines Interfaces unterscheiden.

4.5 Beispiel: JavaDoc von String [35]

```
/**
 * The <code>String</code> class represents character strings. All
 * string literals in Java programs, such as <code>"abc"</code>, are
 * implemented as instances of this class.
 * <p>
 * Because String objects are immutable they can be shared. For example:
 * <p><blockquote><pre>
 *     String str = "abc";
 * </pre></blockquote><p>
...
 * @author Lee Boynton
 * @see    java.lang.Object#toString()
 * @see    java.lang.StringBuffer
 * @since  JDK1.0
 */
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
{
```

```
/**
 * Returns the <code>char</code> value at the
 * specified index. An index ranges from <code>0</code> to
 * <code>length() - 1</code>. The first <code>char</code> value of the
 * sequence is at index <code>0</code>, the next at index <code>1</code>,
 * and so on, as for array indexing.
 *
 * <p>If the <code>char</code> value specified by the index is a
 * <a href="Character.html#unicode">surrogate</a>, the surrogate
 * value is returned.
 *
 * @param     index   the index of the <code>char</code> value.
 * @return    the <code>char</code> value at the specified index of this string.
 *            The first <code>char</code> value is at index <code>0</code>.
 */
public char charAt(int index) {
...
```

4.6 JavaDoc und Pflichtübungen [37]

- Es wird erwartet, dass Sie Ihre Lösungen mit JavaDoc dokumentieren
- alle öffentlichen Methoden und Interfaces müssen dokumentiert werden
- fehlende oder unvollständige JavaDoc führt zu Punktabzügen

Kapitel 5

Versionsverwaltung mit Git

5.1 Das Problem [39]

Wie kann ich …

- … mit mehreren Leuten an einem Projekt entwickeln
- … die Historie der Änderungen der Quellen behalten
- … in der Zeit zurückreisen und alte Stände wieder herstellen
- … herausfinden, welcher Entwickler ein Problem ausgelöst hat
- … verschiedene Versionen der Software entwickeln
- … meine Code ablegen und konsistent per Backup sichern
- … Änderungen aus verschiedenen Quellen zusammenführen

5.2 Die Lösung: Versionsverwaltung (VCS) [40]

- **Versions-Verwaltungssysteme** (Version Control Systems) (**VCS**) verwalten Source Code

  Versions-Verwaltungssysteme
  VCS

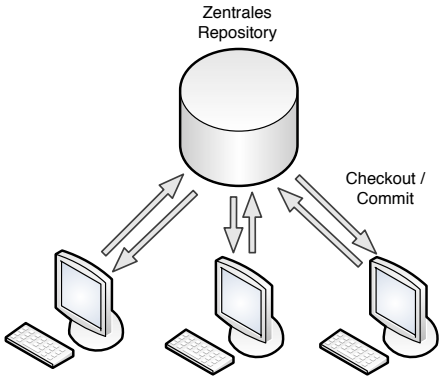
- Eigenschaften
  - *Historie* aller Änderungen
  - *Vergleich* (diff) verschiedener Stände
  - **Abzweigen** (branching) und **Zusammenführen** (merging)

    Abzweigen
    Zusammenführen
  - *Koordinierung* des gemeinsamen Zugriffs auf Source Code
- Man unterscheidet
  - Lokale Versionsverwaltungssysteme
  - Zentrale Versionsverwaltungssysteme
  - Verteilte Versionsverwaltungssysteme

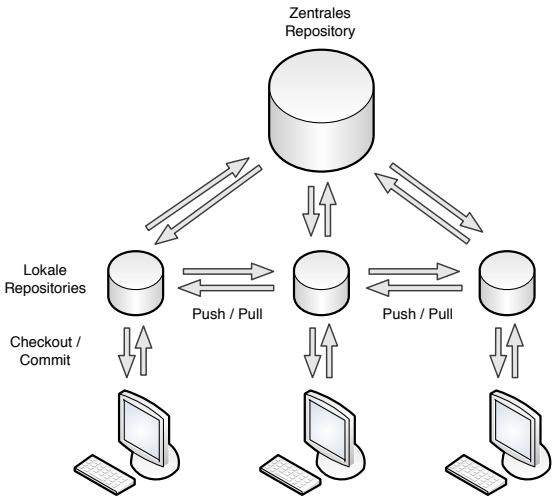
Man spricht auch von **Software Configuration Management Systems** (**SCM**) und meint damit Systeme, die neben der Funktionalität von VCSs noch weitergehende Dienste anbieten. Häufig werden VCS und SCM aber synonym verwendet.

Software Configuration Management Systems
SCM

5.3 Zentrales VCS [41]



5.4 Verteiltes VCS [42]



5.5 Wichtige Begriffe und Konzepte [43]

- **revision** – Version einer Datei im Repository **revision**
- **branch** – Verschiedene Code-Linien, um an parallelen Versionen arbeiten zu können **branch**
- **add** – eine neue lokale Datei dem Repository hinzufügen **add**
- **checkout** – eine Kopie der Datei aus dem Repository holen, um sie lokal zu bearbeiten **checkout**
- **commit** – eine Datei in das Repository zurückstellen **commit**
- **revert** – lokale Änderungen verwerfen und durch den letzten Stand aus dem Repository ersetzen **revert**
- **blame** – anzeigen, wer welche Änderungen an einer Datei gemacht hat **blame**
- **tag** – Revisionen im Repository mit einer Markierung versehen (z. B. „Release 1.0“) **tag**
- **merge** – parallele Änderungen an einer Datei zusammenführen **merge**

- **resolve** – Konflikte beim merge auflösen resolve
- **diff** – Unterschiede zwischen der lokalen und einer zentralen oder zwischen verschiedenen Revisionen einer Datei anzeigen diff
- **clone** – Repository kopieren (nur bei verteilten VCSs) clone
- **push** – Änderungen in ein anderes Repository übertragen (nur bei verteilten VCSs) push
- **pull** – Änderungen aus einem anderen Repository ziehen (nur bei verteilten VCSs) pull

5.6 Zentrale Versionsverwaltungssysteme [45]

Open Source

- *CVS* - Urvater, inzwischen veraltet
- *Subversion* - Populärste zentrale System

Closed Source

- *Visual SourceSafe* - Veraltet, viele Probleme
- *Team Foundation Server* - Nachfolger von Visual Source Safe, Microsoft zentrisch
- *Perforce* - Kostenlos für Open Source oder max. zwei Benutzer:innen

5.7 Verteilte Versionsverwaltungssysteme [46]

Open Source

- *Git* - Linux Kernel Community
- *Mercurial* - Python Community
- *Bazaar* - Ubuntu als Hauptnutzer

Closed Source

- *Bitkeeper*
- *ClearCase*
- *Subversion*

5.8 Git [47]

- Für alle Betriebssysteme verfügbar
  http://git-scm.com
- Eclipse-Integration über eGit
  http://www.eclipse.org/egit
- Zentrales Repository kann über Hoster angelegt werden
  http://www.github.com
- Konzepte sehr ähnlich mit Mercurial (hg)

5.9 Beispiel: Git lokal verwenden [48]

```
> cat ~/.gitconfig
[user]
        name = Thomas Smits
        email = t.smits@th-mannheim.de

> cd project
> git init
Initialized empty Git repository in /Users/thomas/project/.git/

> cat .gitignore
.DS_Store
bin/
.gitignore
.metadata
```

- `.gitconfig` enthält die grundlegende Konfiguration von git
- mit `git init` wird ein neues Verzeichnis für die Versionskontrolle vorbereitet
- `.gitignore` enthält Dateien, die von git ignoriert werden sollen

`git init`

`.gitignore`

```
> git add io
> git status

# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   io/.classpath
#       new file:   io/.project
#       new file:   io/src/de/smits_net/pr2/console/Echo.java.
...
```

Mit `git add` werden neue Dateien hinzugefügt, hier die beiden Ordner `io` und `status`.

`git add`

`git status` zeigt den aktuellen Zustand an, insbesondere welche Dateien noch nicht unter Versionskontrolle stehen oder gegenüber dem Repository verändert wurden.

```
>  git commit -m "Erster Commit des Projektes"
[master (root-commit) b63be32] Erster Commit des Projektes
 40 files changed, 1240 insertions(+), 0 deletions(-)
 create mode 100644 io/.classpath
```

```
 create mode 100644 io/.project
 create mode 100644 io/src/de/smits_net/pr2/console/Echo.java
...

> git status
# On branch master
nothing to commit (working directory clean)

> git log
commit b63be3267c18e3f24db7b36498133a8183a1df49
Author: Thomas Smits <thomsmits@googlemail.com>
Date:   Sat Mar 27 17:28:05 2010 +0100

    Erster Commit des Projektes
```

`git commit` bringt die aktuellen Versionen der Dateien, die mit `git add` hinzugefügt wurden, in das Repository. `git log` zeigt die Historie des Projekts an.

git commit



```
> vi CreateDir.java
> git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes)
#
#       modified:   CreateDir.java
#
no changes added to commit (use "git add" and/or "git commit -a")

> git add CreateDir.java
> git commit -m "Fixed copyright date"
[master d565a0f] Fixed copyright date
 1 files changed, 1 insertions(+), 1 deletions(-)
```



```
> git log
macbook:file thomas$ git log
commit d565a0f757595c83ee265948476ea1ebb461c1ed
Author: Thomas Smits <thomsmits@googlemail.com>
Date:   Sat Mar 27 17:34:10 2010 +0100

    Fixed copyright date

commit b63be3267c18e3f24db7b36498133a8183a1df49
Author: Thomas Smits <thomsmits@googlemail.com>
Date:   Sat Mar 27 17:28:05 2010 +0100

    Erster Commit des Projektes
```

```
> git diff d565a b63be
diff --git a/io/src/de/smits_net/pr2/file/CreateDir.java
     b/io/src/de/smits_net/t
index ef41ca1..05e3ed9 100644
--- a/io/src/de/smits_net/pr2/file/CreateDir.java
+++ b/io/src/de/smits_net/pr2/file/CreateDir.java
@@ -1,5 +1,5 @@
 /*
- * (c) 2010 Thomas Smits
+ * (c) 2009 Thomas Smits
  */
 package de.smits_net.pr2.file;
```

Mit `git diff` kann man sich die Unterschiede zwischen Dateien anzeigen lassen.

5.10 Beispiel: Git verteilt verwenden [54]

```
> git remote add sharedrepo /network/share/pr2/git
> git remote
sharedrepo

> git push sharedrepo
Counting objects: 17, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (9/9), 582 bytes, done.
Total 9 (delta 4), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
To /network/share/pr2/git
   e26823c..91adf73  master -> master
```

`git remote` legt in diesem Beispiel fest, dass mein Repository mit einem anderen verbunden werden soll und das andere als „sharedrepo“ benannt wird. git remote

Mit `git push` wird der lokale Stand des Repositories in das zentrale geschoben. git push

```
> git clone /network/share/pr2/git
Initialized empty Git repository in /Users/thomas/git/.git/
sharedrepo

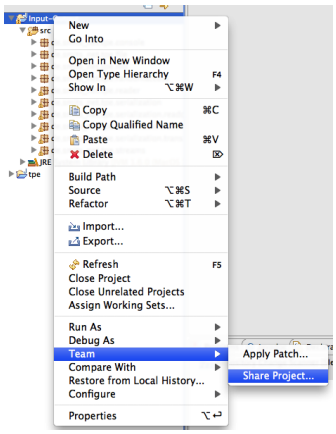
> git pull
remote: Counting objects: 17, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 9 (delta 4), reused 0 (delta 0)
```

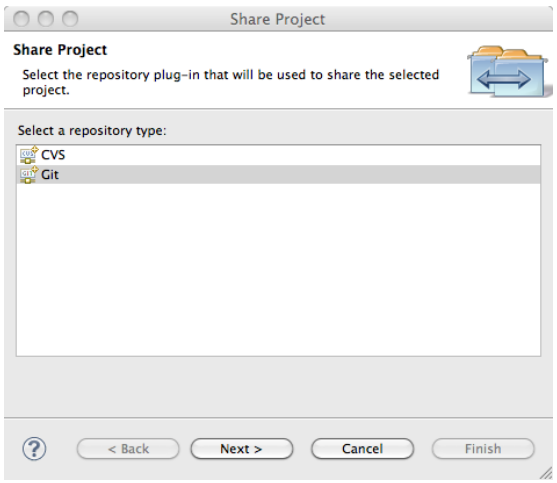
```
Unpacking objects: 100% (9/9), done.
From /Users/thomas/git/ /network/share/pr2/git
   91adf73..f83d24f  master     -> origin/master
Updating 91adf73..f83d24f
Fast-forward
 io/src/de/smits_net/pr2/nio/KopierprogrammNIO.java |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

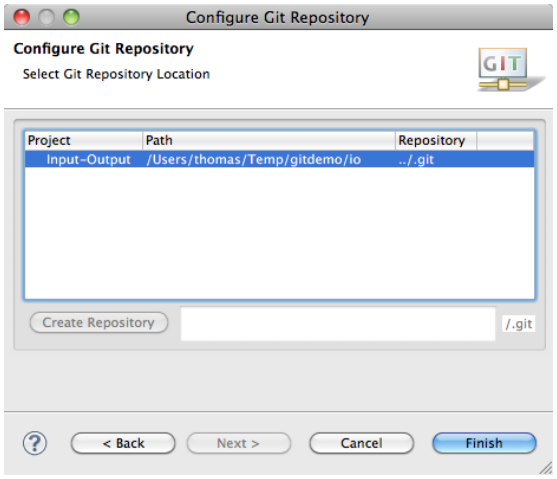
`git clone` kopiert ein Repository und `git pull` importiert die Änderungen aus dem entfernten Repository in mein eigenes Repository.

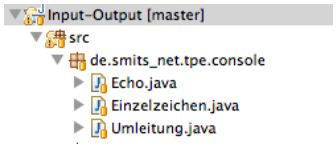
5.11 Git und Eclipse [56]

Git kann direkt aus Eclipse heraus verwendet werden






Configure Git Repository
Configure Git Repository
Select Git Repository Location
GIT
Project
Path
Repository
Input-Output
/Users/thomas/Temp/gitdemo/io
../.git
Create Repository
/.git
< Back
Next >
Cancel
Finish


Input-Output [master]
src
de.smits_net.tpe.console
Echo.java
Einzelzeichen.java
Umleitung.java

Index

Abzweigen, 16
add, 17
Annotationen, 8
Assertion, 8

blame, 17
branch, 17

checkout, 17
clone, 18
Coding-Standard, 3
commit, 17

diff, 18
Don't Repeat Yourself, 5

Header-Datei, 12

JavaDoc, 12
JUnit, 7

magischer Werte, 6
merge, 17
Minimale Sichtbarkeit, 5

Programmierstil, 2
pull, 18
push, 18

resolve, 18
revert, 17
revision, 17

SCM, 16
Software Configuration Management Systems, 16
tag, 17
TDD, 7
Test-Suite, 7
Testgetriebene Entwicklung, 7

Unit-Test, 7

VCS, 16
Versions-Verwaltungssysteme, 16

Zusammenführen, 16
Zusicherung, 8

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Strukturierung



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**
- 1.1 Fragen [5] . . . 1
- 1.2 Begriffe [6] . . . 1
- 1.3 Links zur Java-Insel [7] . . . 1

2 Warum braucht Software Struktur? **2**
- 2.1 Komplex? [9] . . . 2
- 2.2 Zitat [12] . . . 2
- 2.3 Teile und Herrsche (divide and conquer) [13] . . . 3
- 2.4 Beispiel: Fehlende Kapselung [15] . . . 4
- 2.5 Beispiel: Bessere Kapselung [16] . . . 4

3 Konstrukte zur Strukturierung **6**
- 3.1 Elemente zur Strukturierung [18] . . . 6
- 3.2 Klassen [19] . . . 6
- 3.3 Variablen [20] . . . 7
- 3.4 Lebensdauer und Sichtbarkeit [21] . . . 8
- 3.5 Beispiel: Scope und Speicherlayout [22] . . . 8

4 Pakete **10**
- 4.1 Java Pakete [25] . . . 10
- 4.2 Das package Statement [26] . . . 10
- 4.3 Das import Statement [27] . . . 11
- 4.4 Das import static Statement [28] . . . 12
- 4.5 Sichtbarkeit von Klassen [29] . . . 12
- 4.6 Klassen und Dateien [31] . . . 13
- 4.7 Ablage der Dateien [33] . . . 14
- 4.8 Nutzen von Paketen [34] . . . 14
- 4.9 Verwendung von existierenden Paketen [35] . . . 14
- 4.10 Die Datei package-info.java [36] . . . 15
- 4.11 Beispiel: package-info.java [37] . . . 15
- 4.12 Grenzen von Java-Paketen [38] . . . 15

5 Modul-System **17**
- 5.1 Konzept [41] . . . 17

5.2 Umsetzung [43] 18
5.3 Deklaration [44] 18

6 Java-Archive (JAR) 21
6.1 Klassische Programmiersprachen [48] 21
6.2 Compilieren und Linken (statisch) [49] 22
6.3 Compilieren und Linken (dynamisch) [50] 22
6.4 Compilieren und Linken bei Java [51] 22
6.5 Compilieren bei Java [52] 23
6.6 Beispiel: javap [53] 23
6.7 Java ARchive [54] 24
6.8 Weitere Vorteile von Java-Archiven [55] 24
6.9 Erzeugen von JAR-Dateien [56] 25
6.10 Erzeugung einer JAR-Datei [57] 25
6.11 Einbinden von JAR-Dateien [58] 25

Index ii

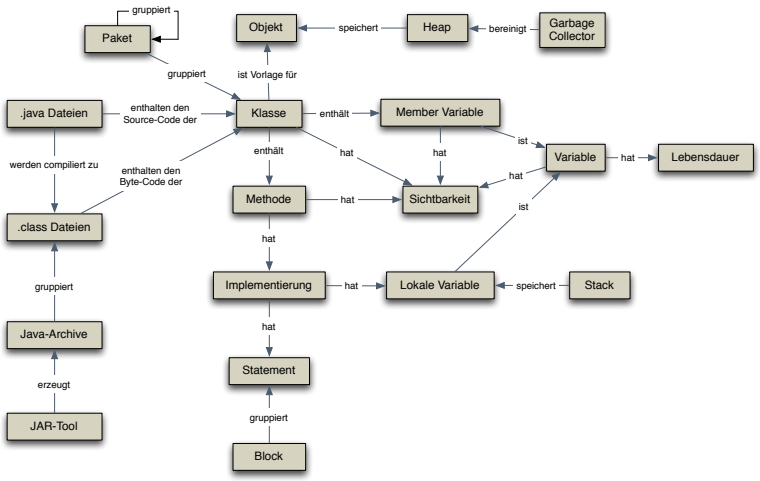
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wenn Programme größer werden, wie behalte ich den Überblick?
- Was hat Julius Caesar dazu zu sagen?
- Wie kann ich dafür sorgen, dass meine Klassen bestimmungsgemäß benutzt werden?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 3.6 Pakete schnüren, Importe und Compilationseinheiten
- 15. Java Platform Module System
- 24.5 Das Archivformat JAR

Kapitel 2

Warum braucht Software Struktur?

2.1 Komplex? [9]

Softwaresysteme gehören zu den komplexesten Gebilden, die Menschen erzeugen

- Boeing 747-400 → 6 Millionen Teile (davon 50% Nieten)
- Windows 2003 Server → 50 Millionen Zeilen Code
- SAP Business Suite → 250 Millionen Zeilen Code
- Menschliches Genom → 3 Milliarden Basen-Paare

Wie können wir trotzdem Kontrolle über die Software behalten?

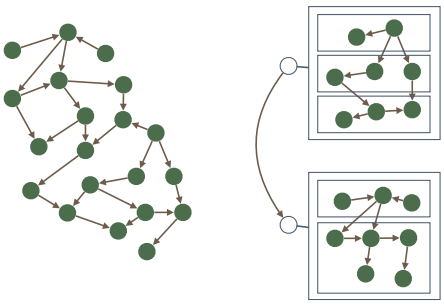
Quellen:

- Boeing Corporation
- Vincent Maraia, The Build Master, Addison-Wesley 2005, Seite xxxiii
- SAP interne Zahlen, bestätig durch SAP-Lab
- Human genome project

2.2 Zitat [12]

> From a bit to a few hundred megabytes, from a microsecond to half an hour of computing confronts us with a completely baffling ratio of 10^9! The programmer is in the unique position that his is the only discipline and profession in which such a gigantic ratio, which totally baffles our imagination, has to be bridged by a single technology. He has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before.
>
> E.W. Dijkstra

2.3 Teile und Herrsche (divide and conquer) [13]



Teilen und Herrschen

Die einzige Möglichkeit, bei großen (und kleinen Softwaresystemen) die Kontrolle zu behalten ist, **Teilen und Herrschen** anzuwenden. Die Software selbst und das Problem müssen in kleine, beherrschbare Teile zerlegt werden und diese Teile müssen über definierte Schnittstellen miteinander interagieren. Diese Zerlegung erfolgt im Allgemeinen auf mehreren Ebenen und mit jeder Ebene wird das Abstraktionsniveau angehoben.

Ein gutes Beispiel für den Ansatz der steigenden Abstraktion ist der Aufbau aller heutigen Computer. Auf der untersten Ebene besteht der Computer aus elektronischen Schaltkreisen, die durch einen sehr maschinennahen Microcode angesteuert werden. Programmen sprechen mit dem Prozessor aber den deutlich abstrakteren Maschinencode (Assembler), der vom Prozessor selbst in Microcode umgesetzt wird. Dieser wird aber im Allgemeinen nicht von Hand geschrieben, sondern von Compilern erzeugt, die eine noch abstraktere Hochsprache (z. B. C++) in den Maschinencode übersetzen können. Bei Java wird sogar noch die Hardware durch die Java Virtual Machine abstrahiert.

Gute Programme zeichnen sich ebenfalls durch eine gute Strukturierung und steigende Abstraktion aus.

Die einzige Möglichkeit größere Softwaresysteme zu konstruieren ist über „teile und herrsche“

- Die Software wird in eigenständige Teile (Module) zerlegt
- Die Schnittstelle zwischen den Modulen wird spezifiziert
- Verwender dürfen nur über die Schnittstelle zugreifen
- Das Innenleben (die Implementierung) des Moduls geht den Verwender nichts an und wird vor ihm versteckt

Dieser Ansatz führt zu

- **Kapselung** (encapsulation) bzw. **Information Hiding**

Kapselung
Information Hiding

Das bahnbrechende Werk zum Thema der Modularisierung und der Datenkapselung stammt von David Parnas: *D.L. Parnas*, On the Criteria to be Used in Decomposing Systems into Modules, Communications of the ACM, December 1972, Volume 15, Number 12

Zum zu versteckenden Innenleben eines Moduls gehören die Daten des Moduls, daher spricht man von **Datenkapselung**. Informationen, die der Verwender eines Moduls nicht braucht,

Datenkapselung

um das Modul zu verwenden werden vor ihm versteckt, weshalb man von Information Hiding spricht.

2.4 Beispiel: Fehlende Kapselung [15]

```
class Datum {
    int tag;
    int monat;
    int jahr;
}

class Verwender {

    void m() {
        Datum d = new Datum();
        d.tag = 32;

        d.tag = 30; d.monat = 2;

        d.tag = 31;
        d.tag++;
    }
}
```

Die Klasse `Datum` verheimlicht seine Implementierung nicht, sondern erlaubt dem Verwender den direkten Zugriff auf die Instanz-Variablen.

Das Problem hierbei ist, dass es damit zum einen unmöglich ist, jemals den Typ der Variablen zu ändern, z. B. das Datum intern durch ein `java.util.Date` abzubilden oder für `tag` ein Feld vom Typ `byte` zu verwenden. Weiterhin ist es `Datum` nicht möglich, Konsistenzprüfungen auf den Daten durchzuführen, da beim Zugriff auf eine Variable kein Code ausgeführt wird. Wie im Beispiel zu sehen, können so ungültige Zustände erzeugt werden. Dies führt dazu, dass die Verwender von Datum sich nicht darauf verlassen können, dass das Datumsobjekt gültige Informationen trägt. Daher müssen die Verwender selbst das Datum auf Richtigkeit prüfen, was eine weitere Verletzung der Kapselung darstellt, da jetzt Programmteile außerhalb des Datentyps Datum über dessen Konsistenz wachen müssen.

2.5 Beispiel: Bessere Kapselung [16]

```
class Datum {

    private int tag;
    private int monat;
    private int jahr;

    public void setTag(int tag) {
```

```
        if (tag > 31) {
            tag = 31;
        }

        this.tag = tag;
    }

    ...
}
```

Natürlich ist der hier gezeigte Source-Code naiv und nicht wirklich gut implementiert. Der Test, ob es sich bei dem Tag um eine gültige Möglichkeit handelt, würde in der Realität sicherlich anders implementiert werden.

Durch die Kapselung der Felder von Datum hat man die Möglichkeit, die interne Darstellung durch eine andere zu ersetzen. Zum Beispiel könnte man statt drei einzelner `int`-Werte auch das Konzept des Unix-Timestamps verwenden und einfach die Millisekunden seit einem bestimmten Zeitpunkt zur Bestimmung des Datums verwenden.

Kapitel 3

Konstrukte zur Strukturierung

3.1 Elemente zur Strukturierung [18]

Elemente zur Strukturierung von Java-Software

- **Anweisungen** (statements) — Anweisungen
- **Blöcke** (blocks) — Blöcke
- **Methoden** (methods) — Methoden
- **Klassen** (classes) — Klassen
- **Pakete** (packages) — Pakete
- **Module** (modules) — Module
- **Java Archive** (JAR) — Java Archive
- **Applikationen** (applications) — Applikationen

Die hier aufgeführte Liste zeigt wieder die verschiedenen Stufen der Abstraktion. Während die Anweisungen das kleinste Element sind, aus dem man ein Java-Programm aufbauen kann, nimmt die Abstraktion von Stufe zu Stufe zu. Blöcke gruppieren mehrere Anweisungen, Methoden gruppieren mehrere Blöcke etc.

3.2 Klassen [19]

In der objektorientierten Programmierung ist eine **Klasse** ein Bauplan oder eine Vorlage, die definiert, welche Attribute (Daten) und Methoden (Funktionen) ein Objekt haben wird, das auf Basis dieser Klasse erstellt wird. — Klasse

Sie beschreibt, welche Eigenschaften und Verhaltensweisen die Objekte haben sollen, die auf Basis dieser Klasse erzeugt werden können. Die Methoden einer Klasse beschreiben, welche Aktionen ein Objekt ausführen kann. Dazu gehören beispielsweise Funktionen zum Verändern der Attribute eines Objekts oder zum Durchführen von Berechnungen.

Klassen sind Baupläne für Objekte, sie beschreiben

- die Daten, die ein Objekt tragen kann (**Variablen**) — Variablen
- das Verhalten, das ein Objekt zeigen kann (**Methoden**) — Methoden

Java-Klassen unterstützen objektorientierte Techniken

- Abstraktion
- Kapselung
- Vererbung
- Polymorphismus

3.3 Variablen [20]

```
class Variablen {
    static int klassenVariable = 10;
    int member = 15;
    int empty;
    void methode(int i) {
        klassenVariable = 11;
        member++; empty++;
        int empty = 5;
        empty++;
        this.empty = 3;
        {
            int k = 5;
        }
        {
            int k = 6;
        }
    }
}
```

Wir unterscheiden:

Klassenvariablen

- **Klassenvariablen** – Sie werden in der Klasse mit dem Schlüsselwort `static` deklariert und existieren unabhängig von den Objekten der Klasse. Pro Klasse (und Classloader) und Klassenvariable gibt es genau einen Wert, unabhängig davon, wie viele Objekte erzeugt wurden. Klassenvariablen werden auch **statische Felder** (static fields) genannt.

statische Felder

Instanzvariablen

Member-Variablen

Feld

- **Instanzvariablen** – Sie werden in der Klasse deklariert und bestimmen die Daten, die die Objekte später tragen werden. Instanzvariablen werden auch **Member-Variablen** oder **Feld** (field) genannt.

Lokale Variablen

Stack-Variablen

automatische Variablen

- **Lokale Variablen** – Sie können innerhalb einer Methode oder auch eines Blocks deklariert werden. Man nennt sie auch **Stack-Variablen** (stack variables) oder **automatische Variablen** (automatic variables).

Lokale Variablen müssen initialisiert werden, bei Klassenvariablen oder Instanzvariablen wird bei fehlender Initialisierung ein Standard-Wert benutzt.

Die Standard-Werte für Klassen und Instanzvariablen sind:

- `byte`, `short`, `int`, `long` → 0

- `float`, `double` → 0.0
- `char` → '\u0000'
- `boolean` → false
- Referenzen → null

3.4 Lebensdauer und Sichtbarkeit [21]

Lebensdauer

- *Klassenvariablen* – während der gesamten Laufzeit des Programms
- *Instanzvariablen* – während der Lebensdauer des Objekts
- *Lokale Variablen* – von der Deklaration bis zum Ende des Blocks oder der Methode, in der sie deklariert wurden

Sichtbarkeit

- *Klassenvariablen*, *Instanzvariablen* – in der gesamten Klasse
- *Lokale Variablen* – von der Deklaration bis zum Ende des Blocks oder der Methode, in der sie deklariert wurden

Korrekterweise muss man sagen, dass Klassenvariablen nicht unbegrenzt leben, sondern beim Laden der Klasse durch den Classloader angelegt werden und beim Entladen der Klasse wieder zerstört werden. Da aber eine Klasse automatisch bei der ersten Verwendung geladen wird und erst entladen und zerstört werden kann, wenn sie nicht mehr in Benutzung ist, fühlt es sich so an, als ob Klassenvariablen während der gesamten Laufzeit des Programms vorhanden sind.

Instanzvariablen werden angelegt, wenn das Objekt mit `new` erzeugt wurde und sie werden zerstört, wenn das Objekt vom Garbage-Collector vernichtet wird.

Lokale Variablen werden beim Betreten der Methode von der Java-VM angelegt und vernichtet, wenn die Methode zurückkehrt. Der Compiler sorgt aber dafür, dass der Zugriff auf die Variablen erst nach der Deklaration erfolgen kann. Technisch wird die Erzeugung und Vernichtung von lokalen Variablen über einen *Stack* realisiert.

Es ist in Java nicht erlaubt, lokale Variablen zu deklarieren, die gleichnamige lokale Variablen eines weiter außen liegenden Blocks verdecken. Das Verdecken von Klassen- oder Instanzvariablen ist dagegen zulässig. Auf verdeckte Instanzvariablen kann man mit der `this`-Referenz zugreifen, auf verdeckte Klassenvariablen mit dem Namen der Klasse. Das Thema Verdecken von Variablen wird später noch einmal aufgegriffen.

Die Sichtbarkeit von Klassen- und Instanzvariablen wird noch von den Schlüsselworten `public`, `protected` und `private` beeinflusst, die später eingeführt werden.

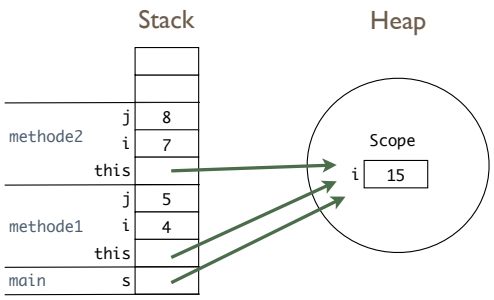
3.5 Beispiel: Scope und Speicherlayout [22]

```
class Scope {
```

```
    int i = 1;
    void methode1() {
      int i = 4; int j = 5;
      this.i = i + j;
      methode2(7);
    }

    void methode2(int i) {
        int j=8;
        this.i = i + j;
    }

    public static void main(String[] args) {
        Scope s = new Scope();
        s.methode1();
    }
}
```



Das Bild zeigt, wie von der Java-VM für jeden Methodenaufruf ein Bereich auf dem Stack angelegt wird. In diesem Bereich finden sich sowohl die lokalen Variablen, als auch die Parameter der Methode. Aus Sicht des Stacks sind lokale Variablen und Parameter identisch. Beide werden beim Betreten der Methode angelegt und beim Verlassen abgeräumt.

Zusätzlich hat jede Methode (solange sie nicht statisch ist) eine Referenz auf das eigene Objekt (`this`).

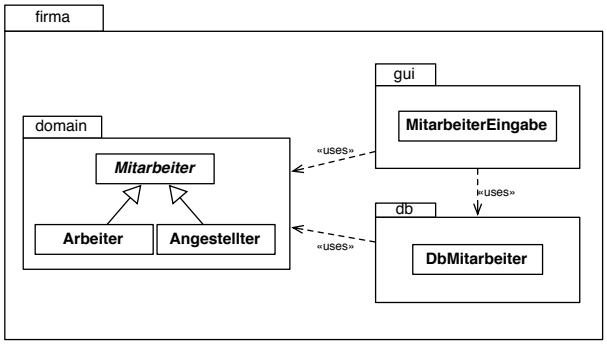
Kapitel 4

Pakete

4.1 Java Pakete [25]

Java Pakete (packages) Java Pakete

- Helfen dabei größere Software-Systeme zu strukturieren
- Können Klassen und selbst wieder Pakete enthalten



Programme können sehr schnell auf viele tausend Klassen anwachsen. Sie würden dann ausgesprochen unübersichtlich, wenn es oberhalb der Klassen keinen weiteren Mechanismus zur Gruppierung gäbe. Genau hier kommen Pakete zur Hilfe. Pakete erlauben, eine beliebige Anzahl von Klassen zu einer Einheit zusammenzufassen, also ein „Paket von Klassen“, die zusammengehören, zu schnüren.

4.2 Das package Statement [26]

In Java dient das package-Statement dazu, Klassen und Interfaces in logische Gruppen zu unterteilen und eine Namensraumhierarchie zu schaffen. Es stellt sicher, dass Klassen und Interfaces innerhalb eines Paketes eindeutige Namen haben, indem ihnen der Paketname vorangestellt wird. Dadurch wird es einfacher, Klassen und Interfaces zu organisieren, zu verwalten und wiederzuverwenden. package

- Klasse ordnet sich selbst durch das package Statement einem Paket zu

- Eine Klasse kann nur einem Paket angehören
- `package` muss als erstes Statement in der Klasse stehen
- In einem Paket kann es jeden Klassennamen nur einmal geben
- Fehlt das `package`, gehört die Klasse zum namenlosen default-Paket
- Paketnamen sind hierarchisch und durch Punkte getrennt

```
package firma.domain;

public class Mitarbeiter { ... }
```

Die Syntax für das *package-Statement* ist

`package <top_pkg_name>[.<sub_pkg_name>]*;`

Kommentare vor dem package Statement sind erlaubt, andere Statements aber nicht.

default-Pakets

Die Verwendung des **default-Pakets**, durch Weglassen des `package`-Statements, ist schlechter Stil und sollte daher in Java-Programmen nicht vorkommen. Sie sollten jede Java-Klasse explizit einem Paket zuordnen.

Die Konvention ist, dass man den Namen des Pakets mit dem umgedrehten Domain-Namen der Firma oder Organisation anfangen lässt, die dieses Paket verantwortet. Da Domain-Namen weltweit eindeutig sind, kann man hierdurch gewährleisten, dass die Paketnamen eindeutig sind. So fangen z. B. alle Paketnamen der Apache Foundation mit `org.apache` an. Beachten Sie, dass in Paketnamen (im Gegensatz zu Domainnamen) kein Bindestrich erlaubt ist. Die Java-Konvention sieht vor, einen Bindestrich im Domainnamen durch einen Underscore (_) im Paketnamen zu ersetzen: aus `hs-mannheim.de` würde dann `de.hs_mannheim`.

4.3 Das import Statement [27]

volle Name

- Der **volle Name** (fully qualified name) einer Klasse besteht aus dem Paket und dem Namen der Klasse, z. B. `java.util.Date`
- Um nicht immer den vollen Namen schreiben zu müssen, kann man dem Compiler kurze Namen über `import` bekannt machen
- Die `import` Statements müssen nach dem `package` Statement aber vor der Klassendeklaration im Quelltext stehen

```
package firma.db;

import firma.domain.Mitarbeiter;
import java.sql.Connection;
import java.util.*;

public class DbMitarbeiter { ... }
```

Die Syntax des `import`-Statements ist

import

```
import <pkg_name>[.<sub_pgk_name>].<class_name>;
```

bzw.

```
import <pkg_name>[.<sub_pkg_name>].*;
```

Mit * kann man alle Namen aus einem Paket importieren. Dies sollte aber vermieden werden, da es den Namensraum der Klasse verschmutzt und durch Lesen der `import`-Statements nicht mehr klar wird, welche Klassen und Interfaces aus dem Paket überhaupt benutzt werden. Da moderne IDEs, wie Eclipse, die Erzeugung der `import`-Statements komplett automatisieren, gibt es heutzutage keinen Grund mehr für die Verwendung von * im Import.

Ob man einen Import per * oder einzeln durchführt, hat keinen Einfluss auf die Compile- oder Laufzeit des Programms, da Compiler und VM intern ohnehin nur mit den **fully qualified name** arbeiten.

fully qualified name

Der Stern bezieht sich immer nur auf die Klassen innerhalb des Paketes nicht aber auf Unterpakete. Man kann damit also keine ganzen Paktehierarchien importieren. Klassen aus Unterpakten werden durch den * somit nicht erfasst.

4.4 Das import static Statement [28]

- Seit Java 5 kann man **static imports** durchführen

static imports

- Macht statische Variablen oder Methoden anderer Klassen bekannt, sodass man sie ohne Angabe des Klassennamens verwenden kann
- Syntax:
  `import static packageName.className.staticMemberName;`

Auch beim `import static` kann man den Stern (*) als Wildcard verwenden. In diesem Fall werden alle statischen Variablen und statischen Methoden der Klasse importiert. Auch hier gibt aber das oben bereits gesagte: man sollte auf Imports per * grundsätzlich verzichten und die statischen Elemente einzeln importieren.

4.5 Sichtbarkeit von Klassen [29]

- Klassen könne zwei mögliche Sichtbarkeiten haben
  - **default** - Nur Klassen innerhalb desselben Pakets dürfen zugreifen

default

  - `public` - Jeder darf zugreifen

public

- Nur wenn die Klasse selbst öffentlich ist, sind die öffentlichen Methoden oder Variablen außen sichtbar



```
package sichtbarkeit;

public class A {
    private int a;
    int b;
```

```
package sichtbarkeit;

class B {
    private int a;
    int b;
```

```
    public int c;                       public int c;
    private void m1() {}                private void m1() {}
    void m2() {}                        void m2() {}
    public void m3() {}                 public void m3() {}
}                                   }
```

Klassen, die nur innerhalb des Paktes sichtbar sind, nennt man auch **paketprivate Klassen** (package private class). Klassen, die mit `public` gekennzeichnet sind, bezeichnet man als **öffentliche Klasse** (public class). paketprivate Klassen öffentliche Klasse

Die Sichtbarkeit von Methoden und Variablen wird noch einmal später genauer erläutert.

In diesem Beispiel gilt Folgendes.

In Klasse `A` ist die Variable `a` nur innerhalb der Klasse, `b` im ganzen Paket und `c` für alle sichtbar. Dasselbe gilt für die Methoden `m1` (nur innerhalb der Klasse), `m2` (im ganzen Paket) und `m3` (überall sichtbar).

Da die Klasse `B` selbst nicht öffentlich ist, sind die Sichtbarkeiten hier: `a` nur innerhalb der Klasse, `b` und `c` nur innerhalb des Paketes. `m1` nur innerhalb der Klasse, `m2` und `m3` nur innerhalb des Pakets.

4.6 Klassen und Dateien [31]

- Jede **öffentliche Klasse** (public class) `ClassName` muss in einer eigenen Datei mit dem Namen `ClassName.java` abgelegt werden öffentliche Klasse

pr2/PublicClass.java

```
package pr2;

public class PublicClass {

}
```

- Mehrere **paketprivate Klassen** (package private class) können in einer Datei abgelegt werden paketprivate Klassen

pr2/PackagePrivateClass.java

```
package pr2;

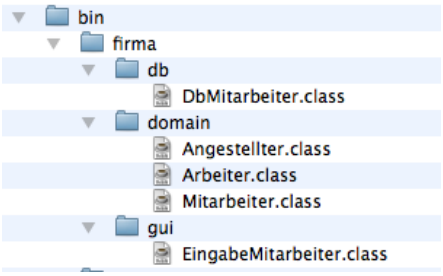
class PackagePrivateClass {
}

class NochEineKlasse {
}

class UndNochEine {
}
```

4.7 Ablage der Dateien [33]

- Kompilierten Java-Klassen (`.class`) werden entsprechend der Paketzugehörigkeit in Verzeichnissen abgelegt `.class`
- Java-Sourcen (`.java`-Dateien) sollten ebenfalls nach Paketen organisiert werden `.java`
- Die IDE (z. B. Eclipse) übernimmt diese Aufgabe automatisch



Java verlangt nicht, dass die Quellen nach Paketen organisiert werden, nur die kompilierten Klassen müssen so abgelegt werden. Aus Gründen der Übersichtlichkeit werden aber die Quellen üblicherweise nach Paketen sortiert. Eclipse macht dies standardmäßig.

4.8 Nutzen von Paketen [34]

- Neben der reinen Strukturierung werden Pakete für das Information Hiding eingesetzt
- Klassen mit Sichtbarkeit auf Paket-Ebene können von außen nicht verwendet werden, daher kann man sie vor externen Verwendern geheim halten

4.9 Verwendung von existierenden Paketen [35]

Die Java VM bringt bereits eine ganze Reihe von Paketen und Klassen mit. Hierbei gilt

- `java.lang` wird immer automatisch importiert, alle anderen Pakete muss man explizit importieren `java.lang`
- `java.*` kann man frei benutzen, Oracle hält die Pakete kompatibel `java.*`
- **Deprecation** dient zum Abkündigen von Funktionen aus den `java.*` Paketen **Deprecation**
- Auf keinen Fall sollte man Klassen aus `sun.*` und `com.sun.*` verwenden

In `java.lang` finden sich die grundlegenden Klassen, die man ohnehin in nahezu jedem Java-Programm benötigt: `Object`, `String`, etc. Aus diesem Grund muss man das Paket nicht importieren, sondern der Compiler macht dies automatisch.

Die Klassen in `sun.*` und `com.sun.*` sind rein interne Implementierungen und unterliegen keinerlei Kompatibiltätsgarantien. Außerdem sind sie nicht Teil der Java-Spezifikation und fehlen auf VMs anderer Hersteller ganz. Wer also solche Klassen verwendet, gibt damit die Plattformunabhängigkeit seiner Java-Programme auf, ein Schritt, den man normalerweise nicht gehen möchte.

4.10 Die Datei package-info.java [36]

- Pakete werden implizit durch die `package`-Deklarationen der Klassen erzeugt
- Man kann eine besondere Datei namens `package-info.java` in den Ordner des Paketes legen
- Sie nimmt auf
  - JavaDoc des Pakets
  - Annotationen die sich auf das ganze Paket beziehen

`package-info.java`

Annotationen (annotations) werden zwar an einigen Stellen in der Vorlesung PR2 benutzt. Die Zeit reicht allerdings nicht aus, um sie grundlegend einzuführen.

Annotationen

Da in Java-Klassennamen keine Bindestriche vorkommen dürfen, besteht keine Konfliktgefahr zwischen der `package-info.java`-Datei und Klassen.

4.11 Beispiel: package-info.java [37]

```java
/**
 * Dokumentation für das Paket packages.
 *
 * <p>
 * Wer dieses Paket verwenden möchte sollte beachten,
 * dass es sich um ein reines Demonstrationspaket handelt.
 * </p>
 *
 * @author Thomas Smits
 * @version 1.0
 */
package pr2.packages;
```

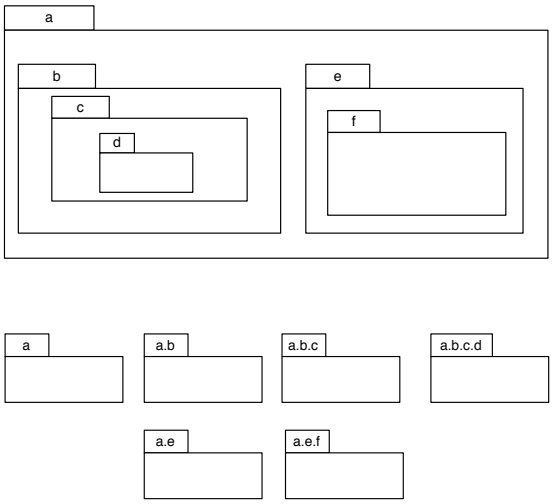
4.12 Grenzen von Java-Paketen [38]

Java-Pakete haben Schwächen

- Pakete bilden keine echte Hierarchie
- Man hat nur die Wahl zwischen package private oder public

Neuere Konzepte versuchen diese Schwächen zu umgehen

- OSGi Bundles
- Module (seit Java 9)



Obwohl die Namen der Java-Pakete und die Ablage in verschachtelten Ordnern den Eindruck erwecken, es handele sich um eine Hierarchie (obere Abbildung), ist es in der Realität anders. Die Hierarchie wird flachgeklopft und es existieren Pakete mit Namen, die einen Punkt enthalten (untere Abbildung). Das Paket `d` im Beispiel befindet sich also gar nicht im Paket `c`, sondern es hat einfach einen Namen, in dem `c` vorkommt, nämlich `a.b.c.d`.

Als Ergebnis haben die Pakete `c` und `d` kein anderes Verhältnis zueinander als die Pakete `d` und `f`.

Module werden im nächsten Kapitel behandelt.

Kapitel 5

Modul-System

5.1 Konzept [41]

Das Java Platform Module System ist eine wichtige Neuerung seit Java 9. Es handelt sich um ein System zur Strukturierung von Java-Anwendungen in wiederverwendbare und unabhängige Module.

Das Modulsystem bietet eine bessere Kontrolle über den Sichtbarkeitsbereich und die Abhängigkeiten von Java-Klassen und ermöglicht eine klare Definition von Schnittstellen und Implementierungen.

Die gesamte Java-Laufzeitumgebung wird in Module unterteilt. Jedes Modul enthält eine Gruppe von zusammengehörigen Klassen und Schnittstellen, die von anderen Modulen wiederverwendet werden können. Jedes Modul hat einen Namen und eine Liste von Abhängigkeiten zu anderen Modulen.

Das Modulsystem definiert auch Regeln für den Zugriff auf Klassen und Schnittstellen innerhalb und zwischen Modulen. Dadurch wird sichergestellt, dass nur die notwendigen Klassen und Schnittstellen zugänglich sind und es wird verhindert, dass unerwünschte Klassen und Funktionen im Code verwendet werden.

Java Platform Module System (**JPMS**) führt **Module** ein.

Java Platform Module System
JPMS
Module

Module

- erlauben, den Zugriff auf Pakete einzuschränken
  ⇒ *Kapselung*
- deklarieren, welche anderen Module sie brauchen
  ⇒ *Klare Abhängigkeiten*, kein Exceptions zur Laufzeit mehr
- erlauben, die Java-Klassenbibliothek selbst zu modularisieren
  ⇒ kleinere Programmpakete

Zugriff auf `public` Typen kann eingeschränkt werden

- auf das eigene Modul
- auf explizit angegebene Module
- auf jeden Verwender

Die letzte Sichtbarkeitsstufe (für jeden Verwender) ist das Standard-Verhalten, wenn man keine Module verwendet. Bis zur Einführung des Modulkonzeptes mit Java 9, konnte jede Klasse jede andere Klasse verwenden, wenn sie `public` deklariert war.

5.2 Umsetzung [43]

Module

- haben eine *Namen*, analog zu Paketen, z. B. `de.hs_mannheim.pr2module`
- werden über die Datei `module-info.java` im Hauptverzeichnis des Moduls gesteuert (**Module Descriptor**)

Module Descriptor

```
module modulename {
    requires MODULE;
    exports PACKAGE;
    ...
}
```

5.3 Deklaration [44]

Direktiven in der `module-info.java`

module-info.java

- `requires MODULE;`
  Definiert eine Abhängigkeit zu einem anderen Modul **(module dependency)**
- `requires static MODULE;`
  Definiert eine (optionale) Abhängigkeit zu einem anderen Modul, die zwingend nur beim Compilieren benötigt wird **(optional dependency)**
- `requires transitive MODULE;`
  Definiert eine Abhängigkeit, die Nutzer von diesem Modul erben **(implied readability)**

Eine transitive Abhängigkeit entsteht, wenn Modul `modA` in seiner Schnittstelle einen Typ (z. B. `T`) von Modul `modX` zurückgibt. Benutzt nun ein anderes Modul `modB` das Modul `modA`, so muss es auch eine Abhängigkeit von Modul `modX` haben, weil es sonst diesen Typ `T` nicht kennen würde, bis es selbst eine Abhängigkeit zu `modX` deklariert. Um diesen Aufwand zu sparen, erklärt `modA` seine Abhängigkeit von `modX` als `transitive`, sodass `modB` automatisch auch von `modX` abhängt, sobald es `modA` benutzt.

- `exports PACKAGE;`
  Exportiert die `public` Typen des Pakets, sodass es andere von außen nutzen können
- `exports PACKAGE to MODULE, ...;`
  Exportiert die `public` Typen des Pakets, sodass es nur die aufgelisteten Module von außen nutzen können
- `provides SERVICE with CLASS;`
  Bietet eine Implementierung des Service mit der gegebenen Klasse an

- `uses SERVICE;`
  Nutzt einen Service
- `opens PACKAGE;`
  Erlaubt Zugriff auf das Paket zur Laufzeit per Reflection
- `opens PACKAGE to MODULE, ...`
  Erlaubt den angegebenen Modulen Zugriff auf das Paket zur Laufzeit per Reflection

Beachten Sie, dass requires ein Modul angibt, exports aber ein Paket.

Services

Services

Das Konzept der **Services** wurde mit Java 9 eingeführt und setzt das Prinzip der Trennung von API und Implementierung (siehe Kapitel zur Objektorientierung) auf der Basis von Modulen um.

Es gibt hier

1. Ein Service Interface
2. Eine oder mehrere Implementierungen des Services

Die Serviceschnittstelle befindet sich in der Regel in einem Serviceschnittstellen-Java-Modul, das nur die Serviceschnittstelle sowie alle mit der Serviceschnittstelle verbundenen Klassen und Interfaces enthält.

Die Service-Implementierungen werden von separaten Java-Modulen bereitgestellt – nicht vom Service-Interface-Modul. Normalerweise enthält ein Java-Modul zur Dienstimplementierung eine einzige Dienstimplementierung.

Ein Java-Modul oder eine Anwendung kann das Serviceschnittstellenmodul verwenden und gegen die Serviceschnittstelle kodieren, ohne genau zu wissen, welches andere Modul die Serviceimplementierung liefert. Die Dienstimplementierung wird zur Laufzeit ermittelt und hängt davon ab, welche Implementierungsmodule beim Start der Anwendung im Java-Modulpfad verfügbar sind.

Service-Schnittstelle

Das Modul mit dem Service-Interface erfordern keine spezielle Deklaration, sondern einfach ein normales Java-Modul.

module-info.java der Service-Schnittstelle

```
module de.hs_mannheim.pr2module.service {
    exports de.hs_mannheim.pr.service;
}
```

Service-Implementierung

Ein Java-Modul, das ein Interface von einem Serviceschnittstellenmodul implementieren möchte, muss:

1. Das Dienstschnittstellenmodul in seinem eigenen Moduldeskriptor per `require` deklarieren.
2. Das Service-Interface mit einer Java-Klasse implementieren.
3. Die Implementierung der Dienstschnittstelle in seinem Moduldeskriptor per `provides` deklarieren.

module-info.java der Service-Implementierung

```
module de.hs_mannheim.pr2module.serviceimpl {
  requires de.hs_mannheim.pr2module.service;

  provides de.hs_mannheim.pr.service.TheSevice with
      de.hs_mannheim.pr2.impl.TheServiceImpl;
}
```

Service-Verwender (Client)

Um den Dienst nutzen zu können, muss das Client-Modul in seinem Moduldeskriptor erklären, dass es den Dienst nutzt.

module-info.java der Service-Implementierung

```
module com.client.theclient {
    requires de.hs_mannheim.pr2module.service;

    uses de.hs_mannheim.pr2module.service.TheService;
}
```

Kapitel 6

Java-Archive (JAR)

6.1 Klassische Programmiersprachen [48]

In klassischen Programmiersprachen besteht die Programmerzeugung aus zwei Schritten

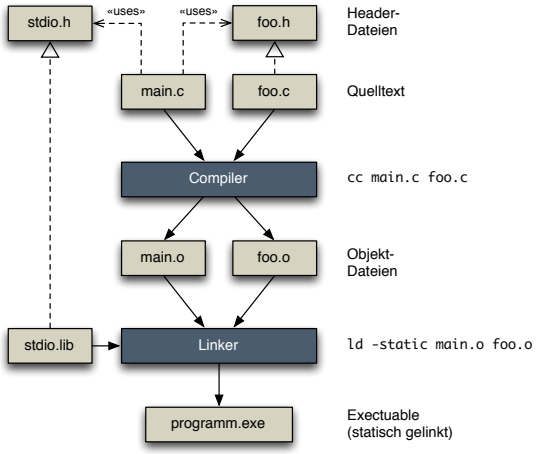
- **Übersetzen** (compile) – die Quelldateien werden einzeln in plattformspezifischen Maschinencode übersetzt — Übersetzen
- **Binden** (link) – alle Teile des Programms werden zu einer einzigen ausführbaren Datei (**Executable**) zusammengebunden — Binden, Executable
  - **statisches Binden** – das erzeugte Executable enthält alle Programmteile und alle Bibliotheken — statisches Binden
  - **dynamisches Binden** – nur Teile des Programms sind im Executable enthalten, andere Teile (meistens Bibliotheken) werden bei Bedarf zur Laufzeit nachgeladen — dynamisches Binden

Klassische Programmiersprachen (C, C++, Modula 2, Pascal) erzeugen ausführbare Dateien, die spezifisch für die jeweilige Plattform sind, d. h. sie enthalten Maschinenbefehle für den Prozessor der Maschine. Um die Programmierung modularisieren zu können, wird die Erzeugung einer solchen Binärdatei in zwei Schritte aufgeteilt:

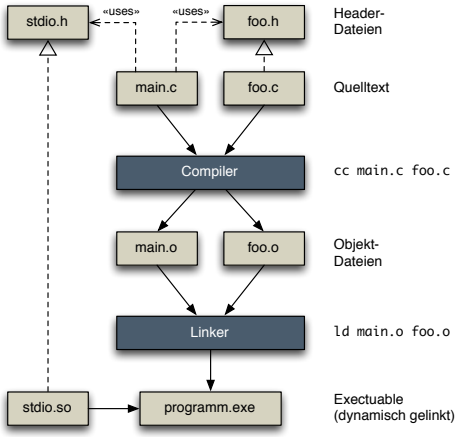
- Im ersten Schritt werden die vorhandenen Quelldateien vom Compiler in Maschinencode für die Plattform übersetzt. Hierbei werden allerdings die Beziehungen zwischen den Quell-Dateien noch nicht aufgelöst, da sie einzeln übersetzt werden. Beziehungen (meist Funktionsaufrufe), die sich von einer auf die andere Quelldatei beziehen werden nur vermerkt aber noch nicht miteinander verbunden. Das Ergebnis eines solchen Schrittes bezeichnet man als **Objektdatei** (object file). — Objektdatei
- In einem zweiten Schritt, sammelt ein weiteres Werkzeug, der **Linker**, alle benötigten Objektdateien zusammen und löst die Querbeziehungen auf, indem er die symbolischen Referenzen durch echte Funktionsaufrufe ersetzt (binden). Als Ergebnis erhält man eine **Programmdatei** (executable). — Linker, Programmdatei

Beim Linken besteht die Möglichkeit entweder alle Objektdateien statisch zu einem einzigen Executable zu binden (statisches Linken) oder aber erst zu Laufzeit einzelne Teile nachzuladen (dynamisches Linken).

6.2 Compilieren und Linken (statisch) [49]



6.3 Compilieren und Linken (dynamisch) [50]



6.4 Compilieren und Linken bei Java [51]

- Java kennt keine Header-Dateien, jede Java-Klasse ist selbstbeschreibend
  - Verwender kann alle Meta-Informationen aus der .class-Datei beziehen
  - `javap` dient dazu, diese Informationen auszugeben
- Java kennt keinen Linker, sondern nur einen Compiler
- Klassen
  - werden von der Java VM dynamisch bei Bedarf geladen
  - werden erst geladen, wenn sie das erste Mal benötigt werden
  - werden erst bei Bedarf von der VM intern gelinkt

- Java VM sucht die Klassen auf dem **Klassenpfad** (classpath) (VM-Option `-cp` oder `-classpath`)

Klassenpfad
-cp
-classpath

Das Übersetzen von Programmen mit Java folgt anderen Prinzipien als das Übersetzen und Linken in klassischen Programmiersprachen.

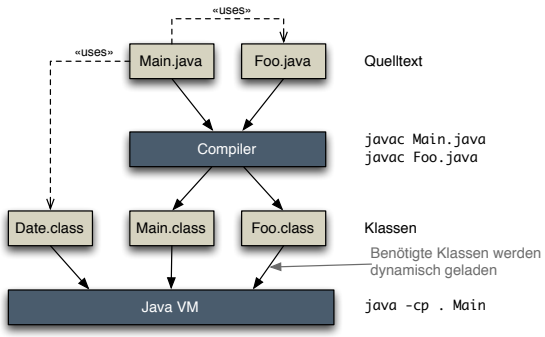
In Java existieren keine Header-Dateien, sondern der Verwender eine Klasse kann alle Informationen, die er benötigt, aus der Klasse selbst beziehen. Java-Klassen sind somit selbstbeschreibend und enthalten sowohl die Definition als auch die Deklaration von Methoden und Variablen. Man kann sich diese Meta-Informationen mit dem Tool `javap` ansehen.

javap

Bei Java existiert kein Linker, d. h. die kompilierten Klassen werden nicht in einem zusätzlichen Schritt zu einer ausführbaren Datei gebunden. An Stelle des Linkers fungiert die Java-VM, die während des Ablauf eines Programms dynamisch alle Klassen nachlädt, die von diesem benötigt werden (dynamic classloading). Natürlich müssen auch im Falle von Java die Beziehungen zwischen den Klassen aufgelöst werden und symbolische Methodenaufrufe durch echte ersetzt werden. Dieses Binden oder Linken wird aber von der VM intern zur Laufzeit durchgeführt und ist daher für den Verwender nicht sichtbar.

Damit die Java-VM die benötigten Klassen finden kann, kann man mithilfe der Kommandozeilenoption `-classpath` festlegen, wo sie überall nach Klassen suchen soll.

6.5 Compilieren bei Java [52]



6.6 Beispiel: javap [53]

```
$ javap java.awt.Point

Compiled from "Point.java"
public class java.awt.Point extends java.awt.geom.Point2D
        implements java.io.Serializable{
    public int x;
    public int y;
    public java.awt.Point();
    public double getX();
```

```
    public double getY();
    public void setLocation(double, double);
    public void move(int, int);
    public void translate(int, int);
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    ...
}
```

Das Beispiel zeigt die Ausgabe des Werkzeugs `javap` für die Klasse `Point`.

6.7 Java ARchive [54]

Java Archive
JAR

Java Archive (**JAR**) – Sammlung von Java-Klassen (und Hilfsdateien) im ZIP-Format gepackt

Vorteile

- komprimiert, weniger Speicherplatzverbrauch
- eine einzelne Datei, mehr Ordnung
- zusammengehörige Klassen können gruppiert werden
- kann direkt ausgeführt werden (`java -jar foo.jar`)
- können genauso auf den Klassenpfad gelegt werden, wie Verzeichnisse mit Klassen

Die große Anzahl von Klassen, die in einem Java-Projekt entstehen, legt es nah, diese in irgendeiner Form zusammenzufassen. Bei einer klassischen Programmiersprache reduziert der Linker die vielen Objektdateien durch das binden zu einigen wenigen Executables und Bibliotheken. In Java gibt es aber keinen Linker, sodass aus jeder Quelldatei mindestens eine .class-Datei entsteht.

Um Java-Klassen zusammenzufassen, verwendet man Java ARchive (JAR). In einem solchen JAR können Java-Klassen und andere Ressourcen, die das Programm zur Laufzeit benötigt, in einer ZIP-Datei verpackt werden. Hierdurch reduziert sich der Speicherplatzverbrauch und die Übersichtlichkeit steigt, wenn man zusammengehörige Klassen in einzelne Archive verpackt.

Die Java-VM kann Klassen nicht nur aus dem Dateisystem als einzelne Dateien, sondern direkt aus dem JAR-Archiv lesen, sodass keine Notwendigkeit besteht, die Archive vor der Ausführung auszupacken. Hierzu gibt man die Java-Archive als Teile des Klassenpfades an.

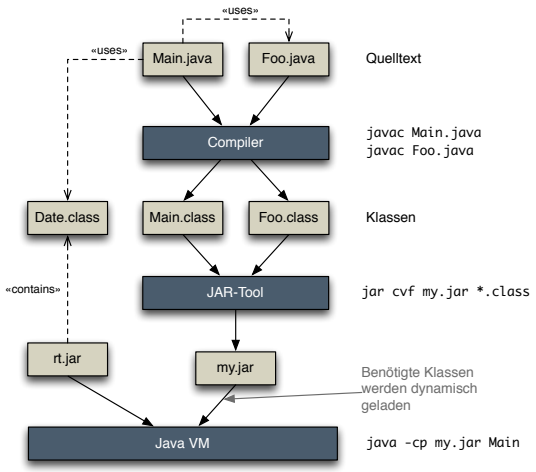
6.8 Weitere Vorteile von Java-Archiven [55]

- *Sicherheit* – Man kann den Inhalt digital signieren und so vor Veränderungen schützen
- *Sealing* – Man kann festlegen, dass alle Klassen eines Paketes in einem JAR enthalten sein müssen
- *Versionierung* – In den Metadaten der JAR-Datei kann man Versionsinformationen für die enthaltenen Klassen ablegen

6.9 Erzeugen von JAR-Dateien [56]

- JAR-Dateien werden mit dem Programm `jar` erzeugt
  `jar -cvf archivname.jar dateien`
- Syntax ist ähnlich zum Unix-Kommando `tar`
- Zusätzlich zu den eingepackten Dateien enthält ein JAR Metadaten im `META-INF`-Verzeichnis

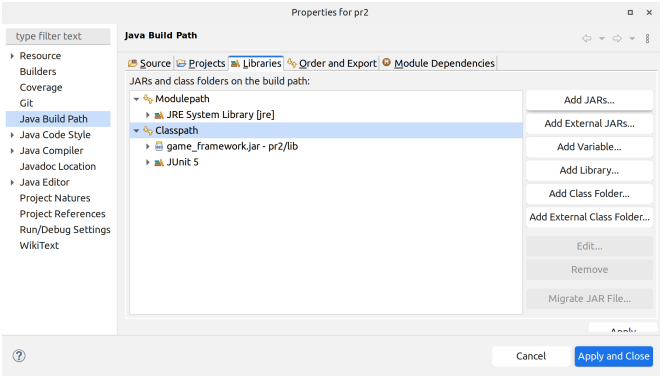
6.10 Erzeugung einer JAR-Datei [57]



6.11 Einbinden von JAR-Dateien [58]

Eigene und fremde JARs können eingebunden werden

- in Eclipse in den Projekteigenschaften unter „Java Build Path“ und Libraries
- auf der Kommandozeile bei Compiler und Java-VM über die Option `-classpath`



Index

Annotationen, 15
Anweisungen, 6
Applikationen, 6
automatische Variablen, 7

Binden, 21
Blöcke, 6

Datenkapselung, 3
default, 12
default-Pakets, 11
Deprecation, 14
dynamisches Binden, 21

Executable, 21

Feld, 7
fully qualified name, 12

Information Hiding, 3
Instanzvariablen, 7

JAR, 24
Java Archive, 6, 24
Java Pakete, 10
Java Platform Module System, 17
JPMS, 17

Kapselung, 3
Klasse, 6
Klassen, 6
Klassenpfad, 23
Klassenvariablen, 7

Linker, 21
Lokale Variablen, 7

Member-Variablen, 7
Methoden, 6
Module, 6, 17
Module Descriptor, 18

Objektdatei, 21

Pakete, 6
paketprivate Klassen, 13
Programmdatei, 21

Services, 19
Stack-Variablen, 7
static imports, 12
statische Felder, 7
statisches Binden, 21

Teilen und Herrschen, 3

Variablen, 6
volle Name, 11

Übersetzen, 21
öffentliche Klasse, 13

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Objektorientierung



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**

1.1 Fragen [5] . . . 1
1.2 Begriffe [6] . . . 1
1.3 Links zur Java-Insel [7] . . . 1

2 Vererbung **3**

2.1 The Royal Baby [9] . . . 3
2.2 Vererbung – Prinz George [10] . . . 3
2.3 Biologische Systematik [12] . . . 4
2.4 Computersoftware [13] . . . 4
2.5 Exkurs: UML [14] . . . 5
2.6 Vererbung: Motivation [15] . . . 5
2.7 Vererbung [17] . . . 6
2.8 Vererbung: Beispiel [20] . . . 8
2.9 Abgrenzung: Vererbung / Aggregation [22] . . . 8
2.10 Zugriffskontrolle (access control) [24] . . . 9

3 Verdecken und Überladen **11**

3.1 Verdecken von Attributen [26] . . . 11
3.2 Überladen von Methoden [28] . . . 12
3.3 Vararg-Methoden (Ellipse) [30] . . . 13
3.4 Beispiel: Ellipse [31] . . . 14
3.5 Namenskonvention für Methoden [32] . . . 15

4 Konstruktoren **16**

4.1 Konstruktoren [34] . . . 16
4.2 Konstruktoren: Beispiel [35] . . . 16
4.3 Der Default-Konstruktor [37] . . . 17
4.4 Überladen von Konstruktoren [38] . . . 17
4.5 Aufruf des Superklassen-Konstruktors [40] . . . 18
4.6 Aufrufhierarchie von Konstruktoren [42] . . . 19
4.7 Copy-Konstruktor [45] . . . 20
4.8 Finalizer sind keine Destruktoren [46] . . . 20
4.9 Unveränderliche Objekte [47] . . . 21

5 Überschreiben von Methoden 23
5.1 Überschreiben von Methoden [51] . . . 23
5.2 Beispiel: Überschreiben von Methoden [52] . . . 23
5.3 Aufruf einer überschriebenen Methode [55] . . . 24
5.4 Überschreiben und Zugriffskontrolle [56] . . . 25
5.5 Überschreiben und Rückgabetyp [57] . . . 25
5.6 Praxistipp: @Override verwenden [58] . . . 26

6 Polymorphie 28
6.1 Polymorphie [60] . . . 28
6.2 Beispiel: Polymorphie [61] . . . 29
6.3 Virtueller Methodenaufruf [62] . . . 29
6.4 Statischer vs. dynamischer Typ [63] . . . 30
6.5 Die Klasse Object [64] . . . 30
6.6 Object als der universelle Variablentyp [65] . . . 31
6.7 Liskovsches Substitutionsprinzip [66] . . . 31
6.8 Casting [68] . . . 32
6.9 Der instanceof Operator [69] . . . 32
6.10 Spätes Binden [72] . . . 33
6.11 Polymorphie und Überschreiben [73] . . . 34
6.12 Polymorphie und Überladen [74] . . . 34
6.13 Typischer Fehler bei Konstruktoren [76] . . . 35

7 Schlüsselwort static 38
7.1 Schlüsselwort static [79] . . . 38
7.2 Statische Methoden und Attribute [80] . . . 38
7.3 Beispiel: Statische Attribute [81] . . . 39
7.4 Beispiel: Statische Methode [83] . . . 40
7.5 Praxis-Tipp: Zugriff auf Klassenvariablen [87] . . . 41
7.6 Statische Blöcke [88] . . . 42
7.7 Das Singleton Pattern [91] . . . 44
7.8 Beispiel: Singleton [92] . . . 44
7.9 Das Static Factory Pattern [94] . . . 45
7.10 Trennung von API und Implementierung [97] . . . 47
7.11 Beispiel: API/IMPL-Trennung [98] . . . 47
7.12 Statische Methoden in Interfaces [101] . . . 48

8 Schlüsselworte final und sealed 49
8.1 Schlüsselwort final [103] . . . 49
8.2 Finale Methoden [104] . . . 49
8.3 Finale Klasse [105] . . . 49
8.4 Finale Variablen [106] . . . 50
8.5 Beispiel: Blank final [110] . . . 51
8.6 Finale Parameter und lokale Variablen [111] . . . 52
8.7 Einkompilieren von Konstanten [112] . . . 53

8.8 Dead Code Elimination [113] 53
8.9 Sealed Classes [117] 54

9 Quiz **57**
9.1 Quiz: Überdecken von Attributen [122] 57
9.2 Quiz: Vererbung [124] 58
9.3 Quiz: Überladen [126] 58
9.4 Quiz: Überschreiben [128] 59
9.5 Quiz: Konstruktoren [130] 60
9.6 Quiz: Statische Methoden [132] 60
9.7 Quiz: Static Initializer [134] 61
9.8 Quiz: Initialisierung [136] 61
9.9 Quiz: Konstruktor [138] 62

Index **iii**

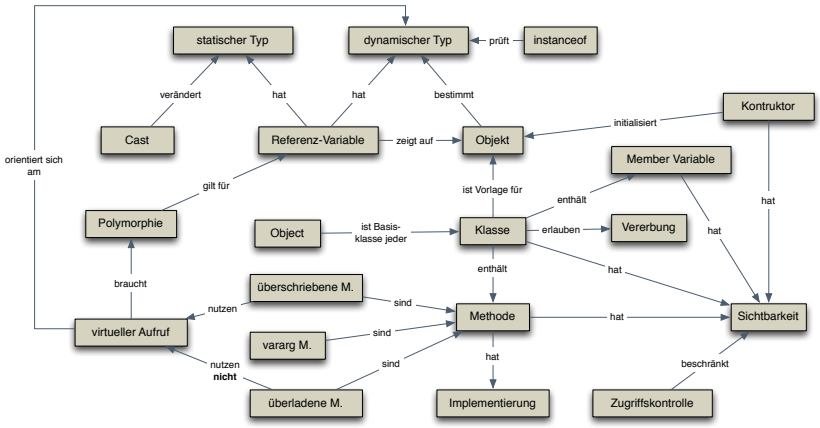
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie vermeidet man, immer alles doppelt und dreifach zu programmieren?
- Wie können Klassen Eigenschaft weitergeben?
- Wie können sich Kinder von ihren Eltern emanzipieren?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 6.3 Eine für alle – statische Methoden und statische Attribute
- 6.4 Konstanten und Aufzählungen
- 7.2 Vererbung
- 7.3 Typen in Hierarchien
- 7.4 Methoden überschreiben

- 7.5 Drum prüfe, wer sich dynamisch bindet

Kapitel 2

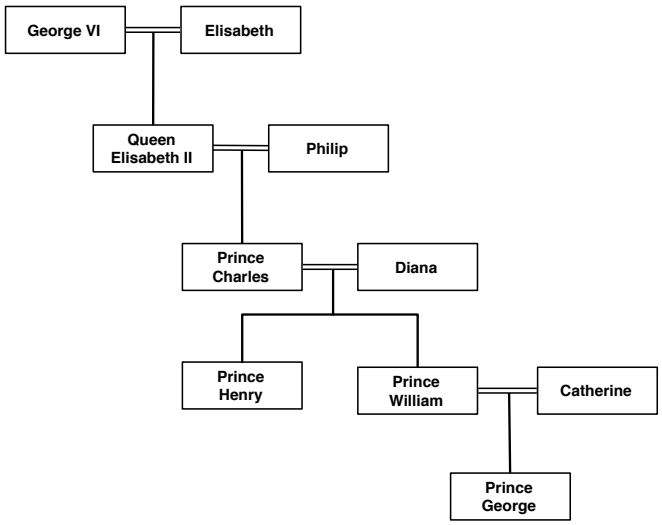
Vererbung

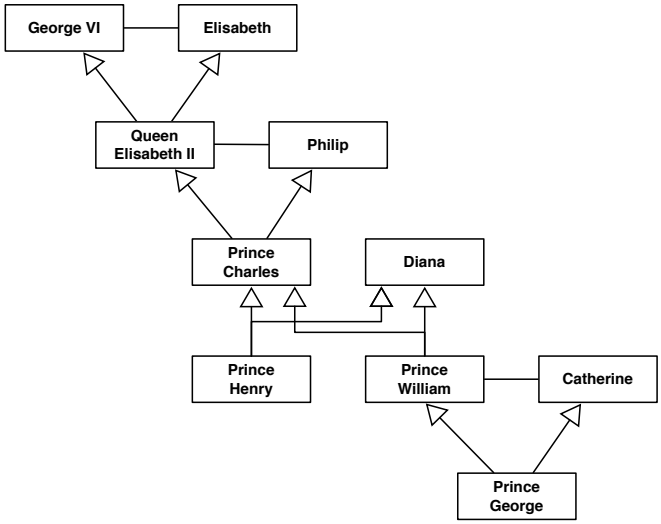
2.1 The Royal Baby [9]

Prince George Alexander Louis of Cambridge (* 22. Juli 2013)

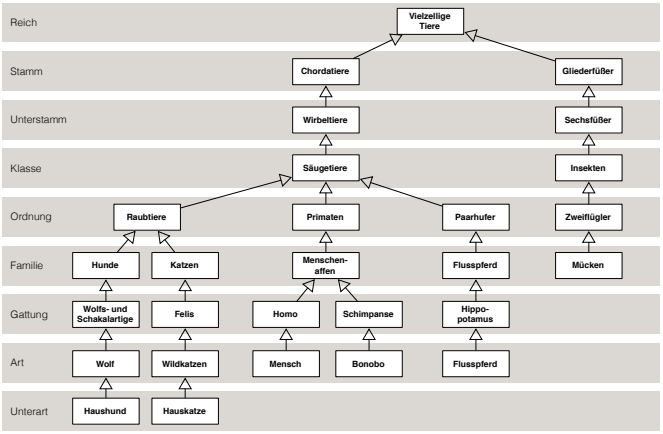


2.2 Vererbung – Prinz George [10]

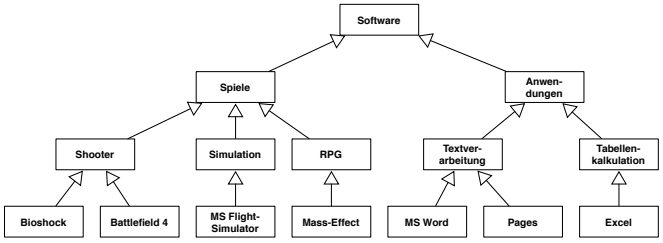




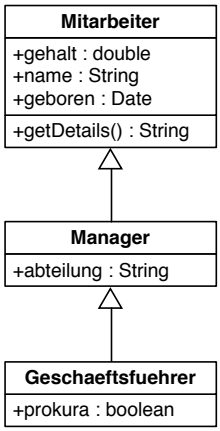
2.3 Biologische Systematik [12]



2.4 Computersoftware [13]



2.5 Exkurs: UML [14]



Jede Klasse wird durch eine Box dargestellt, die aus drei Bereichen besteht:

1. Name der Klasse in Fettdruck
2. Attribute der Klasse mit Sichtbarkeit und Typ
3. Methoden der Klasse mit Parametertypen und Rückgabetypen

Die Typen werden anders als in Java durch einen Doppelpunkt getrennt *hinter* das Attribut, die Methode und den Parameter geschrieben, also `attribut : Typ` bzw. `methodenName(parameterName : Typ): Typ`.

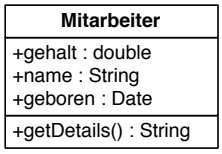
Sichtbarkeit wird später noch detailliert erläutert und ist hier nur der Vollständigkeit halber angegeben.

- \+ → public +
- # → protected #
- ~ → default ~
- - → private -

2.6 Vererbung: Motivation [15]

```
public class Mitarbeiter {
    public String name;
    public double gehalt;
    public Date geboren;

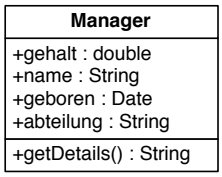
    public String getDetails() {
        return name + ", geb. am " + geboren +
               " hat ein Gehalt von " + gehalt;
    }
}
```



UML Darstellung der Klasse Mitarbeiter

```java
public class Manager {
    public String abteilung;
    public String name;
    public double gehalt;
    public Date geboren;

    public String getDetails() {
        return name + ", geb. am " + geboren +
               " hat ein Gehalt von " + gehalt;
    }
}
```

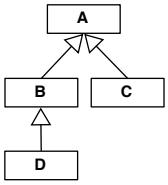


UML Darstellung der Klasse Manager

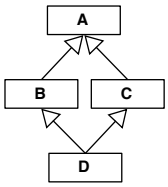
2.7 Vererbung [17]

- **Vererbung** (subclassing) wichtiges Merkmal objektorientierter Sprachen — Vererbung
- Eigenschaften vorhandener Klassen werden auf neue Klassen übertragen
- Man unterscheidet dabei zwischen
  - **Einfachvererbung** (single inheritance) - jede Klasse erbt von maximal einer anderen Klasse — Einfachvererbung
  - **Mehrfachvererbung** (multiple inheritance) - eine Klasse kann von mehr als einer Klasse erben — Mehrfachvererbung
- Java erlaubt grundsätzlich nur Einfachvererbung

Einfachvererbung



Mehrfachvererbung



Man hat sich in Java bewusst dazu entschlossen auf Mehrfachvererbung zu verzichten, da sie zu erheblichen Problemen führen kann. Die Sprache C++, die Mehrfachvererbung unterstützt, ist berühmt/berüchtigt für die Probleme, die entstehen, wenn eine Klasse gleichnamige Attribute und Methoden von mehreren Eltern erbt.

Da es allerdings ein sehr häufiger Anwendungsfall von Mehrfachvererbung in der Beschreibung von Schnittstellen (Interfaces) liegt, hat man für diesen Fall in Java ein gesondertes Konstrukt ersonnen, das später behandelt wird. So ist es gelungen, dass man ohne Mehrfachvererbung sehr gut objektorientiert mit Java programmieren kann.

Neben der Fähigkeit Daten und Methoden zu Klassen bzw. Objekten zusammenzufassen, stellt Vererbung ein weiteres wichtiges Merkmal objektorientierter Sprachen dar. Fehlt die Vererbung spricht man von objektbasierten Sprachen (z. B. ist Visual Basic 6 nur objektbasiert).

Man spricht auch häufig von Ableiten oder Ableitung, wenn man Vererbung meint. Man leitet dann die Klasse B von der Klasse A ab.

- Eine Klasse wird von einer bestehenden Klasse durch Angabe des Schlüsselwortes `extends` abgeleitet
  ```
  public class B extends A {}
  ```
- Die abgeleitete Klasse erbt *alle* Attribute und Methoden der Oberklasse
- Man nennt die Klasse von der abgeleitet wird auch
  - **Superklasse** oder **Oberklasse** (superclass)
  - **Elternklasse** (parent class)

Superklasse
Oberklasse
Elternklasse

- Die Klasse, die ableitet nennt man auch
  - **Subklasse** oder **Unterklasse** (subclass)
  - **Kindklasse** (child class)
  - **Abgeleitete Klasse**

Subklasse
Unterklasse
Kindklasse
Abgeleitete Klasse

Die Aussage, dass die Subklasse alle Attribute und Methoden erbt, wird später noch einmal genauer beleuchtet.

Die Vererbungsbeziehung ist transitiv, d. h. sie bezieht sich nicht nur auf eine direkte Oberklasse, sondern geht über mehrere Hierarchieebenen hinweg. Mit jeder weiteren Ableitung erbt die Subklasse alle bereits an ihre Oberklasse vererbten Attribute und Methoden.

Wenn die Anzahl der Vererbungsebenen zu groß wird, kann die Verständlichkeit des Programms deutlich leiden. Daher sollte man vorsichtig mit Vererbungsbeziehungen umgehen, die 6 oder mehr Ebenen haben.

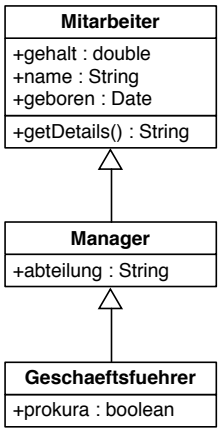
2.8 Vererbung: Beispiel [20]

```java
public class Mitarbeiter {
    public String name;
    public double gehalt;
    public Date geboren;

    public String getDetails() {
        return name + ", geb. am " + geboren +
               " hat ein Gehalt von " + gehalt;
    }
}

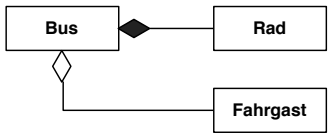
public class Manager extends Mitarbeiter {
    public String abteilung;
}

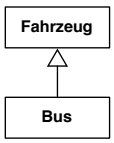
public class Geschaeftsfuehrer extends Manager {
    public boolean prokura = true;
}
```



2.9 Abgrenzung: Vererbung / Aggregation [22]

- *Vererbung* modelliert eine „ist-eine“ Beziehung zwischen Klassen: ein Manager ist ein Mitarbeiter
- *Aggregationen* modellieren eine „hat“ Beziehung zwischen Klassen: ein Manager hat Kinder
- *Kompositionen* sind Aggregationen, bei denen die Elemente nicht getrennt existieren können: ein Manager hat ein Gehirn





Vererbung ist nicht immer das richtige Mittel. Man sollte sie nur einsetzen, wenn man eine „ist-eine“ Beziehung zwischen den Klassen hat, d. h. wenn man sagen kann B ist ein A, z. B. ein Manager ist ein Mitarbeiter, eine Ente ist ein Tier, ein Auto ist ein Fahrzeug etc. Hat man keine „ist-eine“, sondern eine „hat-eine“ Beziehung, so liegt eine Aggregation vor, bei der man nicht mit Vererbung arbeiten sollte. Beispiele für eine Aggregation sind: ein Manager hat Kinder etc.

Man spricht von einer Komposition, wenn die Teile der Aggregation dieselbe Lebensdauer wie das Gesamte haben. So ist z. B. ein Manager nicht ohne Gehirn vorstellbar, sodass Manager und Gehirn eine Komposition bilden.

Wenn man das Konzept der Vererbung zum ersten Mal entdeckt, ist man im Allgemeinen versucht mehr Vererbungsbeziehungen aufzubauen als angemessen wäre. Daher sollte man sich vor jeder Ableitung fragen, ob es sich hier wirklich um eine Vererbungsbeziehung handelt oder eher um etwas anderes. Besonders berücksichtigen sollte man hierbei die Tatsache, dass Vererbung immer ein Verstoß gegen das Prinzip des Information Hiding darstellt, da ja die Subklasse auf eigentlich zu versteckende Eigenschaften Superklasse zugreift.

2.10 Zugriffskontrolle (access control) [24]

- Nicht alle Eigenschaften einer Klasse sollen für jeden Verwender sichtbar sein
- Vier Zugriffsstufen für Variablen und Methoden
  - `private` - Nur Methoden innerhalb derselben Klasse dürfen zugreifen — `private`
  - **default** - Methoden innerhalb desselben Pakets dürfen zugreifen — **default**
  - `protected` - Methoden innerhalb desselben Pakets und Subklassen ausserhalb des Pakets dürfen zugreifen — `protected`
  - `public` - Jeder darf zugreifen — `public`
- Klassen und Interfaces können *default* und `public` sein
- Gutes Design ⇒ minimale Zugriffsrechte für die Verwender

Die hier beschriebene Zugriffskontrolle dient dazu, die bereits beschriebenen Prinzipien der Kapselung und des Information-Hiding bei Java-Klassen umzusetzen.

Die Zugriffsstufe *default* wird häufig als **package-friendly** oder nur package bezeichnet. Für — **package-friendly**

default gibt es kein Schlüsselwort, sondern diese Zugriffsstufe wird durch das Fehlen eines Schlüsselworts angezeigt.

Die höheren Stufen umfassen immer alle niedrigeren Stufen, d. h. eine mit default gekennzeichnete Methode darf natürlich von innerhalb derselben Klasse aufgerufen werden.

Der Zugriff, der über `protected` gewährt wird, bezieht sich auf die geerbten Attribute und Methoden, erlaubt aber keinen generellen Zugriff auf die Felder der Superklasse aus der Subklasse.

```
package pr2.pack1;
public class A {
    protected int i;
    protected void method1() { }
}
```

```
package pr2.pack2;
import pr2.pack1.A;

public class B extends A {
    protected void method2() {
        method1();
        i++;
    }
    public void doIt(A a) {
        a.i++; // FEHLER!!!
    }
}
```

`protected` gewährt einer Subklasse Zugriff auf ererbte Eigenschaften und Methoden einer Superklasse, nicht jedoch auf Interna, die nichts mit der Vererbung zu tun haben. Daher kann im Beispiel die Methode `doIt()` nicht auf das Feld `a` zugreifen, da es sich hier nicht um das geerbte Feld in `B` handelt, sondern um ein Feld in einer übergebenen Instanz von `A`.

Kapitel 3

Verdecken und Überladen

3.1 Verdecken von Attributen [26]

In Java können Klassen Attribute haben, die von Subklassen (abgeleitete Klassen) geerbt werden können. Es ist jedoch möglich, dass eine Subklasse ein Attribut mit demselben Namen wie die Superklasse hat. In diesem Fall spricht man davon, dass das Attribut der Superklasse **verdeckt** wird.

verdeckt

Wenn ein Attribut der Superklasse verdeckt wird, ist das verdeckte Attribut nur in der Superklasse sichtbar und kann nicht direkt von der Subklasse verwendet werden. Stattdessen sieht die Subklasse nur ihr „eigenes“ Attribut.

Wenn eine Methode der Superklasse das verdeckte Attribut verwendet, wird das verdeckte Attribut der Superklasse verwendet und nicht das Attribut der Subklasse. Dies kann zu unerwartetem Verhalten führen, wenn die Subklasse beispielsweise das Attribut anders initialisiert oder es anders verwendet.

Um das verdeckte Attribut der Superklasse trotzdem in der Subklasse zu verwenden, kann das Schlüsselwort `super` verwendet werden.

Verdecken (hiding) von Attributen

Verdecken

- Hat eine Subklasse ein Attribut mit dem gleichen Namen wie die Superklasse, so verdeckt sie das Attribut der Superklasse
- Das neue Attribut wird an alle Subklassen weitergegeben
- Methoden der Superklasse greifen weiterhin auf das alte, Methoden der Subklasse auf das neue Attribut zu

```
public class A {
    public int i = 5;
    public void doIt() {
        System.out.println(i);
    }
}

public class B extends A {
    public String i = "Ein String in B";
```

```
    public void doItNew() {
        System.out.println(i);
        System.out.println(", " + super.i);
    }
}
```

```
new B().doIt() -> 5
new B().doItNew() -> Ein String in B, 5
```

Das Verdecken ist unabhängig vom Typ der Variable und hängt nur vom Namen ab. Es ist sogar möglich ein Attribut durch Überdeckung für alle nachfolgenden Klassen zu entfernen, obwohl es in der Superklasse mit hinreichender Sichtbarkeit deklariert wurde.

```
public class C {
    public int i = 1;
}

public class D extends C {
    private int i; // verdeckt i aus C mit private Sichtbarkeit
}


public class E extends D {
    public void doIt() {
        i++;      // Fehler, i wurde von D mit private verdeckt und
                  // ist daher nicht mehr sichtbar.
        super.i++; // Fehler, da es in D kein sichtbares i gibt
    }
}
```

Wegen der vielen, doch eher überraschenden Effekte des Verdeckens von Feldern, sollte man im Allgemeinen darauf verzichten. Wenn die Variable ohnehin denselben Typ hat, gibt es normalerweise keinen Grund die Variable der Oberklasse zu verdecken. Passt der Typ nicht, ist schwer nachzuvollziehen, warum die Variablen denselben Namen tragen.

3.2 Überladen von Methoden [28]

In Java können Methoden in einer Klasse *überladen* werden, indem mehrere Methoden mit demselben Namen definiert werden, aber mit unterschiedlichen Parametertypen oder -anzahlen. Das Überladen von Methoden ermöglicht es Entwickler:innen, Methoden mit ähnlicher Funktionalität zu gruppieren und gleichzeitig unterschiedliche Parameter zu akzeptieren.

Überladen (overloading) von Methoden

Überladen

- In Java dürfen mehrere Methoden einer Klasse denselben Namen haben, solange sich die Parameterlisten unterscheiden
  - Die *Anzahl* der Parameter ist unterschiedlich und/oder
  - Die *Typen* der Parameter unterscheiden sich
- Der Rückgabetyp der Methode wird nicht beachtet
- Der Compiler wählt die Methode aus, die von der Parameterliste her am besten passt

```
public void gehaltserhoehung(Geschaeftsfuehrer mi) { ... }
public void gehaltserhoehung(Manager mi) { ... }
public void gehaltserhoehung(Mitarbeiter mi) { ... }
```



```
public int average(int a1, int a2) { ... }
public int average(int a1, int a2, int a3) { ... }
public int average(int a1, int a2, int a3, int a4) { ... }
```



In Sprachen ohne Möglichkeit des Überladens bekommt man sehr schnell eine unansehnliche Zahl von Methoden, die alle das gleiche tun aber für verschiedenen Parameterlisten und Parametertypen gedacht sind und daher leicht unterschiedliche Namen tragen: calcDouble, calcFloat, calcInt etc. Durch das Überladen kann man der Methode einen sprechenden Namen geben und die Auswahl der passenden Methode dem Compiler überlassen.

Durch den Einsatz von überladenen Methoden kann man spezialisierte Methoden für Subklassen anbieten und so die Notwendigkeit eines Einsatzes des `instanceof`-Operators und von Casts deutlich reduzieren.

3.3 Vararg-Methoden (Ellipse) [30]

Eine Vararg-Methode, ist eine Methode, die eine variable Anzahl von Argumenten akzeptiert.

Eine Vararg-Parameterliste wird mit drei Punkten `...` nach dem Datentyp definiert. Der Vararg-Parameter muss immer der letzte Parameter einer Methode sein und kann nur einmal in einer Methode verwendet werden.

Die Argumente, die an eine Vararg-Methode übergeben werden, werden in der Methode in Form eines Arrays zur Verfügung gestellt.

Diese Methoden sind nützlich, wenn eine Methode eine flexible Anzahl von Argumenten akzeptieren muss, ohne dass der Aufrufer jedes Mal ein Array erstellen und übergeben muss.

Vararg-Methoden (variable arity method) Vararg-Methoden

- Gelegentlich kann man beim Schreiben der Methode nicht wissen, wie viele Parameter zu erwarten sind

- Durch die Ellipse (...) kann man angeben, dass eine Methode beliebig viele Parameter haben kann
  `public void flexibleMethode(String name, int ... attributes)`
- Die Ellipse muss als letzter Parameter angegeben werden
- Die Methode erhält Zugriff auf die Parameter in Form eines Arrays vom angegebenen Typ (im Beispiel also `int[]`)

Ein klassisches Beispiel für Methoden, die man nur mit Ellipse programmieren kann, sind die `printf()` Methoden von `String` bzw. `PrintWriter`. Bei ihnen hängt die Anzahl der zu erwartenden Parameter von dem Wert des ersten Parameters (des Format-Strings) ab. Ohne die Verwendung der Ellipse müsste man umständlich mit Object-Arrays hantieren. Die Einführung von `printf` war wohl einer der Hauptgründe vararg-Methoden in Java einzuführen.

3.4 Beispiel: Ellipse [31]

```
public class EllipseDemo {

    public void flexibleMethode(String name, int ... attributes) {
        int summe = 0;

        for (int i = 0; i < attributes.length; i++) {
            summe += attributes[i];
        }

        System.out.println(name + " ist " + summe);
    }

    public static void main(String[] args) {
        EllipseDemo demo = new EllipseDemo();
        demo.flexibleMethode("Primsumme bis 13", 2, 3, 5, 7, 11, 13);
        demo.flexibleMethode("Gar nichts");
    }
}
```

Bei der Ellipse handelt es sich um eine reine Compile-Zeit-Konstruktion. D. h. der Compiler setzt sie in eine andere Konstruktion um.

Aus dem folgenden Quellcode

```
public void flexMethod(int ... werte) {
    for (int wert : werte) { /* tu was */ }
}

public void verwender() {
    flexMethod(3, 5, 7, 13, 21, 23);
}
```

wird daher nach dem Compilieren der folgende.

```
public void flexMethod(int[] werte) {
    for (int wert : werte) { /* tu was */ }
}

public void verwender() {
    flexMethod(new int[] { 3, 5, 7, 13, 21, 23 });
}
```

3.5 Namenskonvention für Methoden [32]

In Java hat sich durch die sog. JavaBeans eine Namenskonvention für die Methoden durchgesetzt, die Attribute lesen und setzen

- `setXX` - Setzen des Attributes XX — setXX
- `getXX` - Lesen des Attributes XX — getXX
- `isXX` - Lesen des Attributes XX wenn es `boolean` ist — isXX
- `hasXX` - Lesen des Attributes XX wenn es `boolean` ist — hasXX

Im Gegensatz zu z. B. C# und .NET kennt Java nicht das Konzept der Properties, d. h. von Membern einer Klasse, die zwar mit einer Zuweisung angesprochen werden können (`a.x = 5`), die aber durch methodenähnliche Konstrukte realisiert werden. Daher muss man in Java eine Namenskonvention verwenden und kann leider nicht auf die elegantere Zuweisung zurückgreifen und muss stattdessen eine Methodenaufruf hinschreiben (`a.setX(5)`).

Kapitel 4

Konstruktoren

4.1 Konstruktoren [34]

Konstruktoren (constructors)

- *initialisieren* Objekte bei der Erzeugung
- werden *nicht* vererbt und sind *keine Methoden*
- können überladen werden
- können sich gegenseitig aufrufen und die Konstruktoren der direkten Superklasse rufen

Syntax für den Konstruktor einer Klasse T ist
`<visibility> T(<parameter List>){ .... }`

Der Aufruf anderer Konstruktoren in derselben Klasse erfolgt mit dem Schlüsselwort `this`. Konstruktoren von Superklassen können mit `super` gerufen werden. Es handelt sich aber bei `this` und `super` nicht etwa um Objektreferenzen, sondern um eine spezielle Notation, um andere Konstruktoren zu rufen.

Im Class-File, das der Compiler aus der Klasse erzeugt, finden sich die Konstruktoren nicht mit dem Namen, der im Source verwendet wurde wieder, sondern sie haben den speziellen Namen `<init>`. Auch hieran kann man erkennen, dass Konstruktoren nichts mit Methoden zu tun haben.

4.2 Konstruktoren: Beispiel [35]

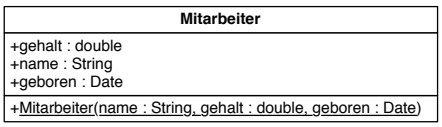
```
public class Mitarbeiter {

    public String name;
    public double gehalt;
    public Date geboren;

    public Mitarbeiter(String name, double gehalt,
           Date geboren) {

       this.name = name;
```

```
        this.gehalt = gehalt;
        this.geboren = geboren;
    }
}
```



Konstruktoren werden in UML im selben Bereich wie Methoden notiert, sie werden aber unterstrichen und heißen genauso wie die Klasse.

4.3 Der Default-Konstruktor [37]

Default-Konstruktor

- **Default-Konstruktor** (default constructor) ⇒ Konstruktor ohne Parameter
- Wenn man in einer Java-Klasse keinen Konstruktor angibt, wird vom Compiler automatisch der Default-Konstruktor mit Sichtbarkeit `public` erzeugt
- Sobald man einen weiteren Konstruktor zur Klasse hinzufügt, wird kein Default-Konstruktor mehr automatisch erzeugt

Der Wegfall des Default-Konstruktors beim Hinzufügen weiterer Konstruktoren ist eine häufige Quelle für Fehler im Programm. Vor allem bei Klassen, die Teil eines APIs sind, kann man hierdurch die Kompatibilität brechen, ohne es zu merken. Daher ist es dringend zu empfehlen, den Default-Konstruktor (so gewünscht) immer explizit hinzuschreiben und die Compiler-Magie nicht zu verwenden.

4.4 Überladen von Konstruktoren [38]

this

- Konstruktoren können wie Methoden überladen werden
- Die Argumentlisten müssen sich unterscheiden
- Mit `this` in der ersten Zeile kann man andere Konstruktoren aufrufen

Beispiele

```
public Mitarbeiter(String name, double gehalt, Date geboren)
public Mitarbeiter(String name, double gehalt)
public Mitarbeiter(String name)
```

```
public class Mitarbeiter {
    public String name;
    public double gehalt;
```

```
    public Date geboren;
    public Mitarbeiter(String name, double gehalt, Date geboren) {
        this.name = name;
        this.gehalt = gehalt;
        this.geboren = geboren;
    }

    public Mitarbeiter(String name, double gehalt) {
        this(name, gehalt, null);
    }

    public Mitarbeiter(String name) {
        this(name, 0.0d, null);
    }
}
```

Mit dem Schlüsselwort `this` kann ein Konstruktor die Arbeit an einen anderen Konstruktor der Klasse delegieren. Der `this`-Aufruf muss das erste Statement im Konstruktor sein, später ist er nicht möglich.

4.5 Aufruf des Superklassen-Konstruktors [40]

`super()`

- Um den Konstruktor der Superklasse aufzurufen, kann man `super()` verwenden
- `super()` muss in der *ersten Zeile* des Konstruktors angegeben werden
- Parameter für den Konstruktor der Elternklasse können angegeben werden
- Macht man keinen expliziten Aufruf, fügt der Compiler einen implizites `super()` ein
  - Gibt es in der Elternklasse keinen Default-Konstruktor bekommt man einen Compiler-Fehler
  - Sind alle Konstruktoren einer Klasse privat, kann man von ihr nicht ableiten und von außen keine Objekte von ihr erzeugen

```
public class Manager extends Mitarbeiter {

    public String abteilung;

    public Manager(String name, double gehalt,
           Date geboren, String abteilung) {

        super(name, gehalt, geboren);
        this.abteilung = abteilung;
    }
}
```

Mit dem Schlüsselwort super kann ein Konstruktor die Arbeit an einen Konstruktor der Superklasse delegieren. Der super-Aufruf muss das erste Statement im Konstruktor sein, später ist er nicht möglich.

4.6 Aufrufhierarchie von Konstruktoren [42]

Möglichkeiten für Konstruktoren

this()

- Man hat explizit ein `this()` geschrieben und delegiert damit an einen anderen Konstruktor derselben Klasse
- Man hat explizit ein `super()` geschrieben und delegiert an einen Konstruktor der Superklasse
- Man hat nichts geschrieben und der Compiler hat automatisch ein `super()` erzeugt
- Mindestens ein Konstruktor enthält kein `this()`, da man andernfalls einen Kreis bilden würde

Was folgt daraus?

```
public class A extends Object {
    protected int i;

    public A(int i) {
        this.i = i;
    }
}

public class B extends A {
    protected int k;
    public B(int k) {
        this(k, 0);
    }
    public B(int k, int i) {
        super(i);
        this.k = k;
    }
}
```

Die Regeln für `super()` und `this()` führen dazu, dass bei jeder Objekterzeugung die gesamte Vererbungshierarchie hinauf alle Konstruktoren bis hin zur Klasse `Object` gerufen werden, d. h. es wird sichergestellt, dass alle Objekte korrekt initialisiert wurden.

- Die Aufrufe der Konstruktoren der Superklassen erfolgen bevor der eigene Konstruktor ausgeführt wird, da `this()` und `super()` als erste Zeile im Konstruktor stehen müssen
- Die geerbten Variablen werden *vor* den eigenen Variablen initialisiert und bevor eigene Anweisungen durchgeführt werden können

Ein verbreiteter Denkfehler ist es anzunehmen, dass durch die Vererbung bei der Erzeugung eines Objekts der Subklasse, Objekte der Superklasse angelegt werden. Dem ist nicht so. Tatsächlich wird nur ein einziges Objekt, nämlich vom Typ der Subklasse, im Speicher alloziert. Dieses enthält Speicherplatz für seine eigenen und alle geerbten Variablen. Die komplexe Aufrufhierarchie der Konstruktoren dient nur dazu, diese Variablen korrekt zu initialisieren, d. h. den Code auszuführen, der zu den Variablen gehört.

4.7 Copy-Konstruktor [45]

Häufig steht man vor der Aufgabe, Objekte mit wiederkehrenden Eigenschaften zu erzeugen oder Objekte zu kreieren, die sich nur in Details voneinander unterscheiden. Hier bietet es sich an, einen speziellen Konstruktor vorzusehen, der Objekte vom eigenen Typ nimmt und daraus ein neues Objekt erzeugt.

Copy-Konstruktor Copy-Konstruktor

- initialisiert das Objekt basierend auf einem anderen Objekt
- ergibt nur als zusätzlicher Konstruktor Sinn

```
public class Mitarbeiter {
    ...

    // Copy-Konstruktor
    public Mitarbeiter(Mitarbeiter other) {
        this(name, gehalt, geboren);
    }
}
```

Mithilfe des Copy-Konstruktors kann man ausgehend von einem Objekt, das als Vorlage dient (**Prototyp-Objekt**), weitere Objekte erzeugen und diese dann nachträglich anpassen. Prototyp-Objekt

4.8 Finalizer sind keine Destruktoren [46]

- In anderen objektorientierten Sprachen (z. B. C++) stellen **Destruktoren** das Gegenstück zu den Konstruktoren dar Destruktoren
- Java kennt wegen der automatischen Speicherverwaltung keine Destruktoren, da die Objekte automatisch zerstört werden
- Java kennt nur die sog. **Finalizer** Finalizer
  - Alle Klassen erben von `Object` eine Methode namens ‘finalize()
  - Sie wird aufgerufen, bevor ein Objekt im Rahmen der Garbage-Collection endgültig zerstört wird
- Man benötigt im Allgemeinen keine Finalizer
- Wegen ihrer unangenehmen Nebenwirkungen sollte man daher sie vermeiden

C++ braucht zwingend das Konzept der Destruktoren, da die Speicherverwaltung von den Programmierer:innen erledigt werden muss. Daher muss er eine Möglichkeit haben, den Zustand seines Objekts zu löschen und allozierten Speicher selbst wieder freigeben zu können. Da in Java die Speicherverwaltung vollkommen automatisch erfolgt, braucht man nicht manuell in die Freigabe des Speichers einzugreifen. In einigen sehr seltenen Fällen möchte man aber sichergehen, dass bestimmte Aufräumarbeiten durchgeführt werden bevor ein Objekt endgültig zerstört wird, z. B. weil eine kritische Ressource (Datenbankverbindung, Datei, Socket etc.) vom Objekt gehalten wird. Für solche Fälle kann man Finalizer verwenden. Ansonsten sollte man sie aber unbedingt vermeiden, da sie einige sehr unerwünschte Eigenschaften haben:

- Es wird nicht garantiert, dass sie überhaupt aufgerufen werden
- Es wird nicht garantiert, wann sie aufgerufen werden
- Finalizer verlangsamen erheblich die Freigabe des Speichers, da die zu finalisierenden Objekte in einer Warteschlange gehalten und einzeln abgearbeitet werden. Das kann zum einen die Gesamtperformance beeinträchtigen und zum anderen auch zu Speicherknappheit führen, wenn der Speicher nicht schnell genug freigegeben wird.

Zusammenfassend kann man sagen: Die Anwendungsfälle für Finalizer sind so selten, dass man sie besser überhaupt nicht verwendet oder aber sehr genau weiß, was man tut.

Details zu Finalizern findet sich in Joshua Block, Effective Java, 2nd Edition, S. 27ff.

4.9 Unveränderliche Objekte [47]

Unveränderliche Objekte

Unveränderliche Objekte (immutable objects)

- Der Zustand des Objekts kann nach der Erzeugung nicht mehr verändert werden
- Nur der Konstruktor setzt den Zustand, keine andere Methode verändert ihn
- Vorteil
  - Es ist irrelevant, ob zwei Referenzen auf zwei gleiche oder ein identisches Objekt zeigen
  - Einige Optimierungen möglich
- Nachteil
  - Handhabung ist komplizierter
  - Performance kann leiden

Das wohl prominenteste Beispiel für ein unveränderliches Objekt in Java ist die Klasse String. Nachdem ein String einmal angelegt wurde, kann er nicht mehr verändert werden. Viele Methoden auf String, wie z. B. `toUpper()` liefern einen neuen String zurück (`s = s.toUpper()`) und verändern niemals den Inhalt des vorhandenen Strings. Darauf muss man beim Umgang mit Strings (und anderen unveränderlichen Objekten) natürlich achten, da ein Aufruf einer Methode deren Rückgabewert ignoriert wird (`s.toUpper()`) einfach wirkungslos bleibt. Weiterhin muss man sich darüber im Klaren sein, dass immutable Objekte in einigen Fällen zu einem signifikanten Performanceproblem führen können, in anderen aber die Performance erheblich steigern können (z. B. über das Fliegengewicht-Pattern).

Durch die Verwendung von unveränderlichen Objekten ist es für den Verwender nicht mehr relevant, ob zwei Referenzen auf zwei gleiche (im Sinne von `equals()`) oder ein und dasselbe Objekt (im Sinne von ==) zeigen. Hierdurch sind viele Optimierungen möglich, die bei veränderbaren Objekten nicht durchgeführt werden könnten. Das Ersetzen zweier gleicher und unveränderlicher Objekte durch ein einziges wird als **interning** bezeichnet.

interning

Für unveränderliche Objekte ergibt ein Copy-Konstruktor nur begrenzt Sinn, weil man die Kopien nicht mehr verändern kann. Deswegen erweitert man die Idee der Getter- und Setter-Methoden um ein weiteres Konzept für unveränderliche Objekte.

Wither-Methoden

Wither-Methoden

- Syntax: `Typ withXXX(Typ xxx)`
- werden bei unveränderlichen Objekten eingesetzt
- liefern ein neues Objekt zurück, das in dem angegeben Attribut geändert wurde

```java
public class City {

    private final String name;
    private final int population;

    public City(String name, int population) {
        this.name = name;
        this.population = population;
    }

    public City withPopulation(int pop) {
        return new City(this.name, pop);
    }

    public City withName(String n) {
        return new City(n, this.population);
    }
}
```

Kapitel 5

Überschreiben von Methoden

5.1 Überschreiben von Methoden [51]

- Häufig will eine Subklasse nicht exakt das Verhalten der Superklasse erben
- Subklassen können daher Methoden der Superklasse durch eigenen Implementierungen ersetzen ⇒ **überschreiben** (overriding) überschreiben
- Die überschriebene Methode hat
  - denselben *Namen*
  - denselben *Rückgabetyp*
  - dieselbe *Parameterliste*
  - aber möglicherweise ein anderes Verhalten als die Methode aus der Superklasse

5.2 Beispiel: Überschreiben von Methoden [52]

```
public class Mitarbeiter {

    public String name;
    public double gehalt;
    public Date geboren;

    public String getDetails() {
        return name + ", geb. am " + geboren +
               " hat ein Gehalt von " + gehalt;
    }
}
```

```
public class Manager extends Mitarbeiter {
    public String abteilung;

    public String getDetails() {
        return name + ", geb. am " + geburtsdatum +
               " hat ein Gehalt von " + gehalt +
```

```
                " und leitet die Abteilung " + abteilung;
    }
}
```

```
public class Geschaeftsfuehrer extends Manager {
    public boolean prokura = true;

    public String getDetails() {
        return name + ", geb. am " + geburtsdatum +
                " hat ein Gehalt von " + gehalt +
                " und leitet die Abteilung " + abteilung +
                " und hat Prokura " + prokura;
    }
}
```

`Manager` und `Geschaeftsfuehrer` `Ã¼berschreiben` `die` getDetails()‘-Methode von Mitarbeiter und passen sie an die jeweiligen Bedürfnisse der Klasse an.

5.3 Aufruf einer überschriebenen Methode [55]

```
public class Manager extends Mitarbeiter {
    public String abteilung;

    public String getDetails() {
        return super.getDetails() +
                " und leitet die Abteilung " + abteilung;
    }
}

public class Geschaeftsfuehrer extends Manager {
    public boolean prokura = true;

    public String getDetails() {
        return super.getDetails() +
            " und hat Prokura " + prokura;
    }
}
```

Auch nachdem die Methode der Oberklasse überschrieben wurde, kann man noch auf sie zugreifen. Hierzu dient das Schlüsselwort `super` gefolgt von dem Namen der Methode aus der Oberklasse, die man rufen möchte.

super

Man kann mit `super` immer nur soweit die Vererbungshierarchie hinaufgehen, bis man das erste Mal auf die Methode mit dem genannten Namen trifft. Es ist nicht möglich darüber

hinaus weiter nach oben zu navigieren, d. h. es gibt keine Möglichkeit aus einer Kette von mehrfach überschriebenen Methoden eine andere als die nächsthöhere aufzurufen.

5.4 Überschreiben und Zugriffskontrolle [56]

Beim Überschreiben einer Methode kann die Sichtbarkeit nur gleich lassen oder erweitern

```
public class A {
    protected void methode() {}
}

public class B extends A {
    public void methode() {}
}
```

```
public class A {
    protected void methode() {}
}

public class B extends A {
    private void methode() {} // FEHLER!!
}
```

Sichtbarkeit wird eingeschränkt: Fehler zur Compile-Zeit.

Der Hintergrund dieser Regel wird sofort klar, wenn man sich mit der Polymorphie beschäftigt, die im nächsten Abschnitt erläutert wird.

5.5 Überschreiben und Rückgabetyp [57]

Beim Überschreiben einer Methode kann man den Rückgabetyp nicht ändern

```
public class A {
    protected void methode() {}
}

public class B extends A {
    public void methode() {}
}
```

```
public class A {
    protected void methode() {}
}
```

```
public class B extends A {
    public int methode() {} // FEHLER!!
}
```

Seit Java 5.0 ist es möglich den Rückgabetyp einer überschriebenen Methode zu ändern solange der neue Typ eine Spezialisierung des überschriebene Typs ist. Man spricht hier von Methoden mit **kovariantem Rückgabetyp** (covariant return type).

kovariantem Rückgabetyp

Folgendes Beispiel ist korrekt für Java 5.0 und neuer, funktioniert aber nicht auf älteren Java-Releases:



```
class A {
}

class B extends A {
}

class C {
    A getFoo() {
        return new A();
    }
}

class D extends C {
    B getFoo() {
        return new B();
    }
}
```

5.6 Praxistipp: @Override verwenden [58]

Überschriebene Methode mit `@Override` annotieren: Der Compiler stellt dann sicher, dass wirklich eine Methode aus der Oberklasse überschrieben wird



```
class A {

    public void method(Object o) {
        // ...
    }
}

class B extends A {

    @Override
    public void method(String o) { // FEHLER!!
        // ...
    }
```

```
}
```

Annotationen sind als Konzept an dieser Stelle zwar noch nicht bekannt, aber dieser Tipp lässt sich recht gut auch ohne detailliertes Wissen über Annotationen verwenden.

Setzt man diesen Trick konsequent ein, verhindert er, dass man aus Versehen eine Methode überlädt anstatt sie zu überschreiben oder in der Oberklasse den Namen oder die Signatur einer Methode ändert und damit die Methode der Unterklasse plötzlich gar keine vorhandene Methode der Superklasse mehr überschreibt. Da diese Art von Fehlern sehr subtil ist und damit schwer zu finden, hilft die `@Override`-Annotation diese zu vermeiden.

In dem hier dargestellten Beispiel stimmt die Signatur der Methode `method` in der Subklasse `B` nicht mit der Signatur der Methode in der Superklasse `A` überein, da in `B` der Parameter von Typ String in `A` aber vom Typ `Object` ist. Hierdurch wird die Methode gar nicht überschrieben, sondern überladen. Die Verwendung der `@Override`-Annotation sorgt dafür, dass ein Compiler-Fehler generiert wird und der Irrtum sofort entdeckt wird.

In vielen IDEs (z. B. Eclipse) kann man konfigurieren, dass die IDE automatisch beim Überschreiben einer Methode die Annotation dazu generiert bzw. warnt, wenn die Entwickler:innen sie bei einer überschriebenen Methode vergessen haben. Diese Feature sollte man auf jeden Fall nutzen.

Kapitel 6

Polymorphie

6.1 Polymorphie [60]

Polymorphie (polymorphism) Polymorphie

- Ein Objekt hat nur eine einzige Form, bestimmt durch seinen Typ
- Eine Variable kann sich **polymorph** (vielgestaltig) verhalten: sie kann auf Objekte verschiedenen Typs zeigen polymorph
  - Eine Variable, vom Typ der Superklasse kann Objekte aller Subklassen referenzieren
  - Es kann aber nur auf die Methoden und Attribute zugegriffen werden, die bereits in der Superklasse deklariert waren

```
Mitarbeiter mi = new Manager();
mi.gehalt += 100.00;

mi.abteilung = "HQX"; // FEHLER!!

Manager ma = new Mitarbeiter(); // FEHLER!!
```

Heide Balzert definiert Polymorphismus wie folgt:

> Ein Name kann Objekte verschiedener Klassen bezeichnen. Jedes Objekt, das durch diesen Namen bezeichnet wird, kann auf die gleichen Botschaften auf seine eigene Art und Weise reagieren. Polymorphismus und spätes Binden sind untrennbar verbunden.
>
> Heide Balzert, Lehrbuch der Objektmodellierung

Der Hintergrund für die Einschränkung der aufrufbaren Methoden und Attribute auf die der Superklasse, liegt an der Typprüfung durch den Compiler. Da es sich bei Java um eine stark typisierte Sprache handelt, überprüft der Java Compiler die Methodenaufrufe oder Attributzugriffe. Der Compiler kann aber nur den Typ der Variablen sehen, nicht aber den Typ des Objekts auf den die Variable zeigt, da dieser erst zur Laufzeit feststeht. Im hier dargestellten Beispiel kann man die Objekttypen zwar aus dem Sourcecode ablesen, was aber normalerweise

nicht möglich ist. Man denke nur an Aufrufe von Methoden, die einen polymorphen Parameter haben.

6.2 Beispiel: Polymorphie [61]

```
Manager ma = new Manager();
Mitarbeiter mi = new Mitarbeiter();

gehaltserhoehung(0.12, ma);
gehaltserhoehung(0.02, mi);

...

public void gehaltserhoehung(double prozente,
       Mitarbeiter mi) {

    mi.gehalt *= 1.0 + prozente;
}
```

Dieses Beispiel zeigt einen sehr häufigen Fall für den Einsatz von Polymorphie: Die Methode `gehaltserhoehung` benötigt nur Daten aus der Klasse Mitarbeiter, um zu funktionieren. Aufgrund der Polymorphie und der Tatsache, dass ein Manager ebenfalls ein Mitarbeiter ist, kann man die Methode aber mit Objekten vom Typ Manager aufrufen. Auch wenn man die Vererbungshierachie um weitere Klassen, wie z. B. `Geschaeftsfuehrer` oder `Vorstand` erweitert, die Methode `gehaltserhoehung` funktioniert weiter problemlos und muss nicht angepasst werden. Dies ist eine ausgesprochene Stärke der objektorientierten Programmierung: ich kann meine Klassen weiter spezialisieren, muss aber die Methoden, die auf Objekten der Superklassen arbeiten, nicht anpassen.

6.3 Virtueller Methodenaufruf [62]

- Polymorphie kann nur funktionieren, wenn bei überschriebenen Methoden die richtige Variante gerufen wird
- Die Entscheidung, welche Methode benutzt wird, muss zur Laufzeit anhand des Objekttyps und nicht zur Compilezeit anhand des Variablentyps erfolgen

```
Mitarbeiter mi = new Mitarbeiter();
Manager ma = new Manager();
System.out.println(mi.getDetails());
System.out.println(ma.getDetails());
```

```
Mitarbeiter mu = new Manager();
System.out.println(mu.getDetails());
```

Man spricht deshalb von einem **virtuellen Methodenaufruf**, weil der Compiler zum Zeitpunkt des Übersetzens nicht weiß, welche Methode letztendlich aufgerufen wird. Er kann daher noch gar meinen „echten“ Methodenaufruf generieren, sondern muss Code erzeugen, der dann zur Laufzeit den Typ feststellt und die richtige Methode ruft. Der vom Compiler erzeugte Methodenaufruf ist also gar kein echter Methodenaufruf, sondern eben nur „virtuell“.

virtuellen Methodenaufruf

6.4 Statischer vs. dynamischer Typ [63]

Eine Variable hat einen

- **statischen Typ** (static type) - Typ mit dem sie deklariert wurde
  - Bestimmt, welche Methoden und Felder sichtbar sind
  - Wird beim Compilieren festgelegt und ändert sich nicht mehr
- **dynamischen Typ** (dynamic type) - Typ des referenzierten Objekts
  - Bestimmt welche Methoden letztendlich aufgerufen werden
  - Kann sich mit jeder Zuweisung ändern

statischen Typ

dynamischen Typ

```
Mitarbeiter mi = new Manager(); // statisch: Mitarbeiter, dynamisch: Manager
mi = new Mitarbeiter();      // statisch: Mitarbeiter, dynamisch: Mitarbeiter
```

Der statische Typ wird häufig **Compile-Zeit-Typ** (compile time type) und der dynamische **Laufzeit-Typ** (runtime type) genannt.

Compile-Zeit-Typ

Laufzeit-Typ

6.5 Die Klasse Object [64]

- Die Klasse `Object` ist die Wurzel aller Vererbungshierarchien
- Jede Java-Klasse erbt direkt oder indirekt von `Object`
- Wenn eine Klasse von keiner anderen erbt, erzeugt der Compiler automatisch ein `extends Object`, d. h. aus
  `public class Klasse {}` wird
  `public class Klasse extends Object {}`
- Die Methoden von `Object` werden später behandelt

`Object`

Die Methoden der Klasse `Object` werden im Folgenden Kapitel behandelt. An dieser Stelle interessieren wir uns nur für die Nutzung von `Object` im Rahmen der Polymorphie.

6.6 Object als der universelle Variablentyp [65]

Da `Object` die Wurzel aller Klassenhierarchien ist, kann eine Variable vom Typ `Object` Referenzen auf Objekte jedes beliebigen Typs aufnehmen

```
Object o = new Manager();
o = new String();

Object[] os = new Object[2];
os[0] = new Manager();
os[1] = new Date();
```

Referenzen vom Typ `Object` sind aufgrund der Polymorphie die universalen Referenzen, da `Object` die Wurzel aller Vererbungshierarchien ist. Sie entsprechen damit ein wenig dem void-Pointer (`void*`) von C, der auch auf jedes beliebige Objekt (im nicht objektorientierten Sinne) auf dem Heap zeigen kann. Natürlich kann man bei der Verwendung von `Object`-Referenzen dann ohne Cast auf keine Felder oder Methoden der dahinter liegenden Objekte zugreifen, da man durch die „Brille“ von `Object` auf die Objekte schaut.

6.7 Liskovsches Substitutionsprinzip [66]

> What is wanted here is something like the following substitution property: If for each object a of type S there is an object b of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when a is substituted for b then S is a subtype of T.
>
> Barbara Liskov, Data Abstraction and Hierarchy, SIGPLAN Notices, 23,5 (May, 1988)



Barbara Liskov

Das **liskovsche Substitutionsprinzip** lautet übersetzt

liskovsche Substitutionsprinzip

> S ist nur dann ein Subtyp von T im Sinne des liskovschen Substitutionsprinzips, wenn überall wo ein Objekt vom Typ T erwartet wird ein Objekt vom Typ S benutzt werden kann ohne dass sich das Programm anders verhält

Man kann mit Vererbung natürlich problemlos Vererbungshierarchien bauen, die das liskovsche Substitutionsprinzip verletzen, z. B. indem man von einer Klasse Rechteck die Klasse Quadrat ableitet und dann in der Klasse Quadrat einen Fehler wirft, wenn jemand versucht

unterschiedliche Werte für die beiden Seitenlängen anzugeben. Obwohl es sich in diesem Fall um ein vollkommen korrektes Konstrukt handelt, das vom Compiler akzeptiert wird, verletzt es das liskovsche Substitutionsprinzip mit der Folge, dass Quadrat kein Subtyp im liskovschen Sinne von Rechteck ist.

Wenn man Klassen und Vererbungshierarchien so baut, dass das liskovsche Substitutionsprinzip eingehalten wird, sind die Programme robuster und besser zu verstehen. Andernfalls vererbt man zwar Implementierung und Daten, aber es entsteht keine vollständige ist-eine Beziehung.

6.8 Casting [68]

Cast

- Durch einen **Cast** gibt man explizit den dynamischen Typ an. Es handelt sich um einen Operator. Syntax: `(TYP)`
- Methoden und Attribute werden entsprechend des angegebenen Typs sichtbar
- Casts werden erst zur Laufzeit überprüft (⇒ Laufzeitfehler möglich)

```
Mitarbeiter mi = new Manager();

// Down-Cast
Manager ma = (Manager) mi;
ma.abteilung = "Buchhaltung";

// Down-Cast in kompakter Form
((Manager) mi).abteilung = "Buchhaltung";
```

6.9 Der instanceof Operator [69]

instanceof

- Manchmal muss man zur Laufzeit feststellen können, welchen Typ ein Objekt hat, z. B. um sicher casten zu können
- Der `instanceof`-Operator prüft den Typ einer Variable
  - `a instanceof T` liefert `true`, wenn das Objekt mit der Referenz a vom Typ T oder wenn das Objekt hinter a vom Typ einer Subklasse von T ist
  - `null instanceof T` ist immer `false`
  - wenn `a instanceof T` `true` liefert, wird ein Cast von a auf T (`(T) a`) niemals fehlschlagen (keine Ausnahme werfen)

```
Mitarbeiter mi = new Manager();

(mi instanceof Mitarbeiter) --> true
(mi instanceof Manager) --> true
(mi instanceof Geschaeftsfuehrer) --> false
(null instanceof Mitarbeiter) --> false
```

Der `instanceof` Operator liefert für alle Subklassen `true` zurück, d. h. man kann ihn nicht verwenden, wenn man sicher sein will, dass ein Objekt exakt von einem bestimmten Typ ist. Hierzu gibt es aber im Rahmen von Reflection Möglichkeiten, die aber in der Realität nicht häufig benutzt werden.

Wichtig ist anzumerken, dass man `instanceof` mit Nullreferenzen verwenden darf. Hier ist dann per Definition das Ergebnis immer `false`. Anstatt `if (s != null && s instanceof String)` kann man kürzer `if (s instanceof String)` schreiben.

```
public void automatischeErhoehung(Mitarbeiter mi) {

    if (mi instanceof Geschaeftsfuehrer) {

        Geschaeftsfuehrer g = (Geschaeftsfuehrer) mi;

        if (g.prokura) {
            mi.gehalt = mi.gehalt * (1 + 0.2);
        } else {
            mi.gehalt = mi.gehalt * (1 + 0.1);
        }
    } else if (mi instanceof Manager) {
        mi.gehalt = mi.gehalt * (1 + 0.05);
    } else {
        mi.gehalt = mi.gehalt * (1 + 0.002);
    }
}
```

Dieses Beispiel zeigt, wie man mithilfe des instanceof-Operators abhängig vom dynamischen Typs des übergebenen Objekts, unterschiedliche Aktionen durchführen kann und wie man einen Cast (hier auf `Geschaeftsfuehrer`) durch eine entsprechende Prüfung mit `instanceof` schützen kann.

Es ist aber anzumerken, dass längere Ketten von if/else if-Konstrukten mit `instanceof` als schlechter Programmierstil gelten und daher vermieden werden sollten. Der Grund hierfür ist, dass ein solcher Stil anfällig für Fehler ist, die durch Veränderungen an der Vererbungshierarchie entstehen:

Wenn eine weitere Klasse hinzugefügt wird, muss man gewährleisten, dass ein passendes `if` eingebaut wird.

Weiterhin müssen die Tests immer in der richtigen Reihenfolge durchgeführt werden, nämlich von unten nach oben die Vererbungshierarchie herauf. Macht man hier einen Fehler, greift schon ein weniger spezieller Fall, als erwartet.

6.10 Spätes Binden [72]

Spätes Binden (late binding) Spätes Binden

- Damit der virtueller Methodenaufruf funktioniert, darf die Entscheidung, welche Methode gerufen wird, erst zur Laufzeit getroffen werden

- Der Compiler kann diese Entscheidung nicht treffen, sondern muss Code generieren, der dies zur Laufzeit anhand des dynamischen Typs der Variable durchführt

Das späte Binden wird häufig als dynamisches Binden bezeichnet.

Bei Java generiert der Compiler, im Gegensatz zu anderen Sprachen, allerdings gar keinen besonderen Code für das späte Binden, sondern erzeugt nur einen besonderen Byte-Code namens `invokevirtual` und überlässt dann die Magie des dynamischen Bindens der Java Virtual Machine.

6.11 Polymorphie und Überschreiben [73]

Beim Überschreiben von Methoden

- darf der *Rückgabetyp nicht verändert* werden
- darf die *Sichtbarkeit nicht eingeschränkt* werden
- dürfen nur Exceptions weggelassen oder durch speziellere Exceptions (Subklassen) ersetzt werden

Warum?

- Der Verwender darf nicht überrascht werden
- Was mit der Superklasse geht, muss auch mit der Subklasse funktionieren

Der Grund für diese Regeln liegt in der Polymorphie, da der Verwender einer Variablen nicht damit rechnen kann, dass das Objekt ein anderes Verhalten zeigt, als der statische Typ der Variable suggeriert. Daher darf man eine Klasse, die man ableitet, nur so verändern, dass der Verwender nicht merken kann, ob er mit der Super- oder Subklasse hantiert. Sogar der instanceof-Operator ist so konstruiert, dass er bei Subklassen noch `true` liefert. Andernfalls würde man nämlich den Verwender überraschen.

Wie bereits erwähnt, sind diese technischen Anforderungen relativ schwach verglichen mit dem likovschen Substitutionsprinzip, das eine deutlich höhere Anforderung an das Verhalten von überschriebenen Methoden und abgeleiteten Klassen stellt.

Im Prinzip handelt es sich bei diesen Regeln nur um die technische Ausgestaltung der folgenden Bedingung:

- Die **Vorbedingung** (precondition) einer Methode einer abgeleiteten Klasse darf nur schwächer als die der Superklasse sein — Vorbedingung
- Die **Nachbedingung** (postcondition) einer Methode einer abgeleiteten Klasse darf nur stärker sein als die der Superklasse — Nachbedingung

6.12 Polymorphie und Überladen [74]

- Polymorphie wirkt sich *nicht* auf überladene Methoden aus
- Compiler wählt die überladene Methode aus, nicht die Laufzeit



```
public class UeberladenUndPolymorphie {

    public void method(A a) {
        System.out.println("method(A a) gerufen");
    }

    public void method(B b) {
        System.out.println("method(B b) gerufen");
    }
 }
```

```
UeberladenUndPolymorphie u = new UeberladenUndPolymorphie();

A a1 = new A();
A a2 = new B();

u.method(a1);
u.method(a2);
```



```
Ausgabe
method(A a) gerufen
method(A a) gerufen
```


Die Annahme, dass Polymorphie sich auf überladene Methoden bezieht, ist falsch und trotzdem häufig anzutreffen.

Bei überladenen Methoden sucht der Compiler beim Übersetzen der Klassen nach einer Methode, deren Parameterliste und Parametertypen am besten zu dem gefunden Methodenaufruf passen und generiert dann einen entsprechenden Bytecode, der die gefundene Methode aufruft. Der Compiler kann aber nur die statischen Typen sehen, d. h. die Typen der Referenzvariablen, nicht aber die dynamischen Typen (d. h. die Typen der Objekte, die von den Variablen referenziert werden). Somit ist es nicht möglich bei überladenen Methoden ein Verhalten ähnlich der Polymorphie zu erhalten, das sich an den dynamischen Typen orientiert. Die Auswahl der überladenen Methode findet immer zur Übersetzungszeit statt.

6.13 Typischer Fehler bei Konstruktoren [76]

Auf keinen Fall sollte man Methoden, die überschrieben werden können, vom Konstruktor aus aufrufen

```
public class A {

    protected String name;
```

```
    public A() {
        name = getDetails();
    }

    protected String getDetails() {
        return "Hans";
    }
}
```

```
public class B extends A {

    protected String info = "";

    protected String getDetails() {
        return name + info.length();
    }
}
```

Hier führt jetzt `new B();` zu einer `NullPointerException`

Unter „Methoden, die überschrieben werden können“ fallen alle nicht nicht-finalen (wird später noch erläutert) Methoden, die mindestens default-Sichtbarkeit haben. Diese Regel greift also nicht für private Methoden oder für Methoden, die eine größere Sichtbarkeit als private haben aber final sind. Aus diesem Grund, sollte man beim Aufruf von Setter-Methoden aus Konstruktoren vorsichtig sein, da diese sehr häufig überschreibbar sind.

Das Problem lässt sich relativ einfach lösen, indem man die Funktionalität aus der Methode `getDetails` in eine private Methode auslagert und diese dann aus dem Konstruktor und der `getDetails`-Methode benutzt. Die Lösung sähe dann wie folgt aus:

```
public class A {

    protected String name;

    public A() {
        name = getDetailsInternal();
    }

    private final String getDetailsInternal() {
        return "Hans";
    }

    protected String getDetails() {
        return getDetailsInternal();
    }
}
```

```
public class B extends A {

    protected String info = "";

    public B() {
    }

    protected String getDetails() {
        return name + info.length();
    }

    public static void main(String[] args) {
        B b = new B();
        System.out.println(b.getDetails());
    }
}
```

Kapitel 7

Schlüsselwort static

7.1 Schlüsselwort static [79]

Man kann das Schlüsselwort `static` verwenden für

static

- Variablen
- Methoden
- Innere Klassen

Es bedeutet, dass

- eine Methode losgelöst von einer Instanz existiert
- eine Variable zur Klasse und nicht zu den Instanzen gehört
- eine innere Klasse ohne dazugehöriges Objekt existieren kann

7.2 Statische Methoden und Attribute [80]

Statische Methoden (static methods)

Statische Methoden

- können ohne Objekt direkt auf der Klasse aufgerufen werden
- können selbst nur auf statische Methoden und statische Attribute zugreifen
- können nicht überschrieben, sondern nur überlagert werden

Statische Attribute (static attributes)

Statische Attribute

- existieren nur einmal pro Klasse
- können direkt über die Klasse angesprochen werden
- können *überlagert* (*verdeckt*) werden

Da eine Klasse in Java durchaus mehrmals vorkommen kann, nämlich dann, wenn sie durch verschiedene Classloader geladen wurde, muss man korrekterweise sagen, dass statische Attribute pro Instanz der Klasse (d. h. pro Classloader und Klasse) nur einmal vorkommen. Dieses Detail wollen wir hier aber ausblenden und sagen, dass sie nur einmal existieren.

Das Überschreiben von Methoden ist eine Eigenschaft der objektorientierten Programmierung, die den virtuellen Methodenaufruf, spätes Binden und Polymorphie voraussetzt. Für statische

Variablen und Methoden existieren diese Mechanismen aber in Java nicht, d. h. obwohl man glaubt die Methode überschrieben zu haben, hat man sie tatsächlich einfach nur verdeckt bzw. überlagert. Leider erlaubt der Java-Compiler das Verdecken von statischen Methoden und Variablen, obwohl hier ein großes Gefahrenpotential einem geringen Nutzen gegenübersteht.

```
public class A {

    public static void tuWas() {
        System.out.println("A:tuWas()");
    }
}

public class B extends A {

    public static void tuWas() {
        System.out.println("B:tuWas()");
    }
}

public class Test {

    public static void main(String[] args) {
        A a1 = new A();
        B b = new B();
        A a2 = new B();
        a1.tuWas();
        b.tuWas();
        a2.tuWas();
    }
}
```

Die Ausgabe ist:

```
A:tuWas()
B:tuWas()
A:tuWas()
```

7.3 Beispiel: Statische Attribute [81]

```
public class Ticket {

    public static int zaehler = 0;

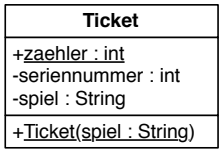
    private int serienNummer;
    private String spiel;

    public Ticket(String spiel) {
```

```
        zaehler++;
        serienNummer = zaehler;
        this.spiel = spiel;
    }
}
```

Im vorliegenden Beispiel ist die Variable `zaehler` statisch deklariert und existiert nur einmal pro Klasse. Die Variablen `serienNummer` und `spiel` sind aber normale Instanzvariablen, die im Objekt gespeichert werden.

Immer wenn ein neues Objekt angelegt wird, wird die statische Variable `zaehler` hochgezählt und der `serienNummer` zugewiesen. So bekommt jedes Objekt vom Typ `Ticket` eine eindeutige Seriennummer.



In der UML werden statische Attribute und Methoden durch Unterstreichung gekennzeichnet.

7.4 Beispiel: Statische Methode [83]

```
public class Ticket {

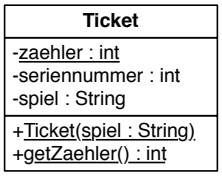
    private static int zaehler = 0;

    private int serienNummer;
    private String spiel;

    public Ticket(String spiel) {
        zaehler++;
        serienNummer = zaehler;
        this.spiel = spiel;
    }

    public static int getZaehler() {
        return zaehler;
    }
}
```

Dieses Beispiel ist analog zum vorhergehenden. Der Unterschied besteht nur darin, dass der Wert von `zaehler` nicht mehr direkt von außen gelesen werden kann, sondern über die statische Methode `getZaehler()`.




```java
public class TestTicket {

    public static void main(String[] args) {

        Ticket t1 = new Ticket("Dortmund-Schalke");
        System.out.println(Ticket.getZaehler());

        Ticket t2 = new Ticket("Hoffenheim-Leverkusen");
        System.out.println(Ticket.getZaehler());
    }
}
```

```
1
2
```

7.5 Praxis-Tipp: Zugriff auf Klassenvariablen [87]

Verwenden Sie beim Zugriff auf statische Variablen und Methoden *immer* den Klassennamen und *nie* eine Objektreferenz

```java
Variablen.klassenVariable = 10;

// Schlechter Stil!!
Variablen v = new Variablen();
v.klassenVariable = 15;
```

Der Java-Compiler bindet die Klassenvariable und die Methode an den Typ der Objektreferenz und nicht an den Typ des Objekts (*statische Bindung*)

Die statische Bindung kann zu sehr unerwarteten Effekten führen, da statische Variablen von Subklassen überdeckt werden können. Das folgende Beispiel zeigt, warum man auf keinen Fall über Objekt-Referenzen auf Klassen-Variablen zugreifen sollte:

```java
class A {
    static int wert = 5;
}
```

```
class B extends A {
    static int wert = 6;
}

public class C {
    void method(A a) {
        System.out.println(a.wert);
    }

    public static void main(String[] args) {
        new C().method(new B());
    }
}
```

Das Beispiel liefert als Ausgabe 5, da der Java-Compiler den Zugriff auf das statische Feld `wert` anhand des Typs des Parameters `a` der Methode `method()` bestimmt und nicht anhand des Typs des Objekts, dass `method()` übergeben wird.

7.6 Statische Blöcke [88]

Statische Blöcke (static initializer) Statische Blöcke

- existieren außerhalb einer Methode
- dienen der Initialisierung der Klasse
- werden exakt in der Reihenfolge ihrer Deklaration ausgeführt
- werden exakt einmal beim Laden der Klasse ausgeführt
- können an beliebiger Stelle in der Klasse stehen
- können mehrfach pro Klasse vorkommen
- haben Zugriff auf die statischen Attribute und statischen Methoden der Klasse

Statische Blöcke kann man sich als Klassen-Konstruktoren vorstellen. In der erzeugten Klasse werden sie in einer speziellen Methode namens `<clinit>` gesammelt. Im Gegensatz zu normalen Konstruktoren können sie aber keine Parameter bekommen und werden automatisch beim Laden der Klasse durch die Java-Virtual-Machine ausgeführt.

Innerhalb statischer Blöcke können Fehler auftreten. Da der statische Block aber nicht explizit aufgerufen wird, sondern implizit durch den Classloader, hat man als Programmierer:in keine Möglichkeit Fehler außerhalb eines statischen Blocks explizit zu fangen. Daher muss man besonders defensiv in statischen Blöcken programmieren. Kommt es trotzdem zu einem Fehler, so wird das Laden der Klasse abgebrochen und sie steht dann dem Programm nicht zur Verfügung, was im Allgemeinen zu einem Programmabsturz führt. Das folgende Beispiel zeigt, was passiert, wenn eine Ausnahme in einem statischen Block auftritt.

```
public class Crash {

    static {
        int x = 5 / 0;
```

```
    }

    public static int add(int a, int b) {
        return a + b;
    }
}
```

```
public class CrashUser {

    public static void main(String[] args) {
        System.out.println(Crash.add(5, 5));
    }
}
```

Der Fehler im statischen Block führt dazu, dass die Klasse nicht geladen werden kann und damit der `main`-Methode nicht zur Verfügung steht. Bei dem Versuch, die `add`-Methode zu nutzen, wird eine Ausnahme geworfen.

```
Exception in thread "main" java.lang.ExceptionInInitializerError
  at pr2.oo2.statisch.initializer.CrashUser.main(CrashUser.java:7)
Caused by: java.lang.ArithmeticException: / by zero
  at pr2.oo2.statisch.initializer.Crash.<clinit>(Crash.java:7)
  ... 1 more
```

```
public class Ticket {
    public static int zaehler;
    private int serienNummer;
    private String spiel;
    static {
        zaehler = Integer.parseInt(System.getProperty("zaehler_start"));
    }

    public Ticket(String spiel) {
        zaehler++;
        serienNummer = zaehler;
        this.spiel = spiel;
    }

    static {
        System.out.println("Ticket initialisiert: zaehler=" + getZaehler());
    }
}
```

Das Beispiel liest den Anfangswert für den Seriennummern-Zähler aus einer System-Property, die der VM beim Start durch die -D-Option übergeben werden kann.

Der Code ist insofern naiv, als dass er nicht damit umgehen kann, dass die Property eventuell nicht gesetzt wurde, sondern in diesem Fall mit einer `NullPointerException` abstürzt.

```
public class TestTicket {

    public static void main(String[] args) {

        Ticket t1 = new Ticket("Dortmund-Schalke");
        System.out.println(Ticket.getZaehler());

        Ticket t2 = new Ticket("Hoffenheim-Leverkusen");
        System.out.println(Ticket.getZaehler());
    }
}
```

```
>java -Dzaehler_start=100 TestTicket
Ticket initialisiert: zaehler=100
101
102
```

7.7 Das Singleton Pattern [91]

Zweck des **Singletons** Singletons

- es soll genau *eine einzige Instanz* eines Objektes geben
- es soll einen einzigen, globalen Zugriffspunkt auf dieses Objekt geben

Das Singleton erschwert aber auch

- Threadsicherheit
- Rekonfiguration
- Multi-user, multi-client Betrieb

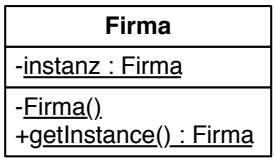
7.8 Beispiel: Singleton [92]

```
public class Firma {
    private static final Firma INSTANZ = new Firma();

    public static Firma getInstance() {
        return INSTANZ;
    }

    private Firma() {
        // Keine Instanzen dieser Klasse!
    }
```

```
}
```




```
public class Firma {

    private static final Firma instanz;

    static {
        instanz = new Firma();
    }

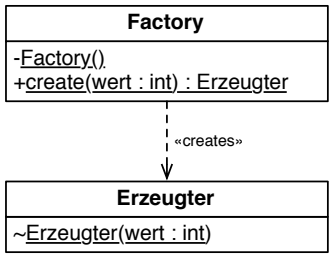
    public static Firma getInstance() {
        return instanz;
    }

    private Firma() {
        // Keine Instanzen dieser Klasse!
    }
}
```

7.9 Das Static Factory Pattern [94]

Zweck der **Static Factory** Static Factory

- Kontrollierte Erzeugung von Objekten, ohne dass der Verwender den konkreten Typ kennt
- Verwendung anstatt Konstruktoren
- Klassen können sich selbst erzeugen



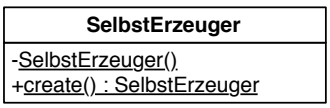

```
public class Erzeugter {
```

```
    int wert;

    Erzeugter(int wert) {
        this.wert = wert;
    }
}
```

```
public class Factory {

    public static Erzeugter create(int wert) {
        return new Erzeugter(wert);
    }
}
```



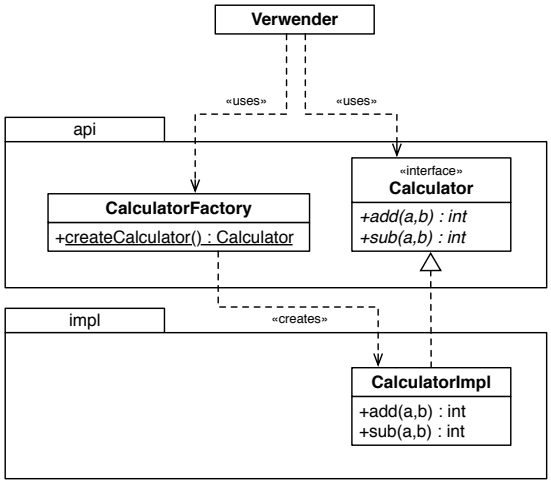
Selbsterzeugung

```
public class SelbstErzeuger {

    private SelbstErzeuger(int wert) {
        this.wert = wert;
    }

    public static SelbstErzeuger create() {
        return new SelbstErzeuger();
    }
}
```

7.10 Trennung von API und Implementierung [97]



Ein gutes Design trennt **API** (**Application Programming Interface**) und Implementierung. Unter API versteht man die Schnittstelle zu einem Softwaremodul. Eine solche Trennung können wir mit den bisher vorgestellten Techniken und Entwurfsmustern (Pattern) herbeiführen. Die Schnittstelle zur Funktionalität wird durch ein Interface beschrieben, das zusammen mit einer statischen Fabrik in einem speziellen API-Paket liegt. Die Implementierung des Interfaces (Funktionalität des API) liegt in einem anderen Paket und kann nur über die Factory erzeugt werden.

API
Application Programming Interface

Mit diesem Design hat man ein Maximum an Flexibilität:

- Der Verwender kennt nur eine Schnittstelle, weiß aber nicht welche Klasse diese implementiert.
- Die genaue Implementierung ist komplett vor dem Verwender verborgen und kann jederzeit geändert werden. Noch nicht einmal die Klasse, die das API implementiert, ist dem Verwender, dank der statischen Fabrik, bekannt.

7.11 Beispiel: API/IMPL-Trennung [98]

```
public interface Calculator {
    public abstract int add(int a, int b);
    public abstract int sub(int a, int b);
}
```

```
public class CalculatorImpl implements Calculator {

    public int add(int a, int b) {
        return a + b;
    }
```

```
    public int sub(int a, int b) {
        return a - b;
    }
}
```

```
public class CalculatorFactory {
    private CalculatorFactory() {}

    public static Calculator createCalculator() {
        return new CalculatorImpl();
    }
}
```



```
public class Verwender {

    public static void main(String[] args) {

        Calculator calculator = CalculatorFactory.createCalculator();

        int ergebnis = calculator.add(1, 5);

        System.out.println(ergebnis);
    }
}
```


7.12 Statische Methoden in Interfaces [101]

- Seit Java 8 dürfen Interfaces statische Methoden enthalten
- Unterscheiden sich nicht von statischen Methoden in Klassen (siehe oben)

```
public interface Rechner {

    public int apply(int a, int b);

    public static int rechne(Rechner r, int a, int b) {
        return r.apply(a, b);
    }
}
```


Statische Methoden an Interfaces sind sinnvoll, wenn man Hilfsfunktionen anbieten möchte, die eng mit dem Interface verbunden sind. Bisher musste man immer eine weitere Klasse angeben, die die statischen Methoden enthielt. Jetzt können diese direkt am Interface verbleiben.

Kapitel 8

Schlüsselworte final und sealed

8.1 Schlüsselwort final [103]

Das Schlüsselwort `final` hat mehrere Bedeutungen

`final`

- Eine **finale Klasse** kann keine Subklassen haben — **finale Klasse**
- Eine **finale Methode** kann nicht überschrieben werden — **finale Methode**
- Eine **finale Variable** kann nicht mehr verändert werden — **finale Variable**

8.2 Finale Methoden [104]

```
public class A {
    public void methode() {}
    public final void finalMethode() {}
}
```



```
public class B extends A {

    @Override
    public void methode() {}

    @Override
    public void finalMethode() {} // FEHLER!!
}
```



8.3 Finale Klasse [105]

```
public final class A {

}
```



```
public class B extends A { // FEHLER!!

}
```



8.4 Finale Variablen [106]

- Finale Variablen können nicht mehr verändert werden
  - Finale Variablen *primitiver Typen* sind echte Konstanten
  - Finale Variablen *unveränderlicher Typen* verhalten sich ähnlich wie Konstanten
  - Finale Variablen *veränderlicher Typen* schützen das Objekt nicht vor Änderungen (sollte vermieden werden)
- Die Zuweisung an eine final Variable muss nicht in der Deklaration erfolgen (**blank final**)

blank final

  - Der Compiler stellt sicher, dass die Variable auf allen möglichen Pfaden vor der ersten Verwendung genau einmal gesetzt wird

Eine finale Variable kann nachträglich nicht mehr geändert werden, wohl aber das Objekt auf das sie zeigt. `final` sorgt nur dafür, dass die Variable nicht mehr auf der linken Seite einer Zuweisung auftauchen kann.

Somit verhält sich final anders als das `const` in C++, bei dem man tatsächlich durch konsequentes Durchziehen des `const`-Modifiers durch alle Methoden, Parameter, Variablen und Klassen dafür sorgen kann, dass als `const` deklarierte Objekte nicht mehr verändert werden können. In Java hat man von diesem Ansatz wegen seiner Komplexität Abstand genommen. Die Folge ist aber, dass es keinen wirklichen Schutz vor unbeabsichtigter Veränderung von Objekten gibt, den man bei der Verwendung noch hinzufügen kann. Entweder ein Objekt ist von einem unveränderlichen Typ oder nicht.

```
public class Konstanten {

    // echte Konstante
    public static final double PI = 3.14159265;

    // echte Konstante
    public static final double NA = 6.0221415E23;

    // String ist immutable, verhält sich wie String-Konstante
    public static final String XML_HEADER =
           "<?xml version=\"1.0\"?>";

    // nur die Variable, nicht aber das Array sind geschützt
    public static final int[] PRIMZAHLEN = { 2, 3, 5, 7, 11, 13 };
```



```
}
```

In diesem Beispiel sind `PI` und `NA` echte Konstanten, weil sie von einem primitivem Typ sind. `XML_HEADER` verhält sich wie eine Konstante, weil das String-Objekt unveränderlich ist.

Die Variable `PRIMZAHLEN` ist zwar als `final` deklariert, dies bezieht sich aber nur auf die Variable selbst, nicht aber auf das dahinterliegende Array. Insofern kann man die Daten noch verändern, wie das folgende Beispiel zeigt.

```
public class Konstanten {
    public static final int[] PRIMZAHLEN =
           { 2, 3, 5, 7, 11, 13, 17, 19 };
}

public class Konstantenverwender {

    public static void main(String[] args) {
        System.out.println(Arrays.toString(Konstanten.PRIMZAHLEN));
        Konstanten.PRIMZAHLEN[2] = 8;
        System.out.println(Arrays.toString(Konstanten.PRIMZAHLEN));
    }
}
```

```
[2, 3, 5, 7, 11, 13, 17, 19]
[2, 3, 8, 7, 11, 13, 17, 19]
```

Das Beispiel zeigt, dass es sinnlos ist, `final` im Zusammenhang mit veränderlichen (mutable) Typen zu verwenden. Obwohl hier die Variable `PRIMZAHLEN` als `final` deklariert ist, kann man das Array auf das sie verweist problemlos modifizieren. In diesen Fällen sollte man das `final` am besten ganz weglassen, um keine falsche Sicherheit zu suggerieren.

Wenn man Listen von Objekten benötigt, die unveränderlich sind, bietet das Collection-API entsprechende Möglichkeiten, die später noch beleuchtet werden. Arrays sind hier auf jeden Fall ungeeignet.

8.5 Beispiel: Blank final [110]

```
public class Kunde {

    private final int kundenNummer;

    public Kunde(int kundenNummer) {
        this.kundenNummer = kundenNummer;
    }
```

```
}
```

Das Beispiel zeigt, dass man das Setzen einer finalen Variablen durchaus herauszögern kann. Wichtig ist hier, dass die Variable auf jeden Fall genau einmal zugewiesen wird und danach nicht mehr. Der Compiler stellt dies sicher, indem er alle Ausführungspfade untersucht. Da hier die Variable `kundenNummer` im Konstruktor gesetzt wird, darf man sie als `final` deklarieren. Nach der Initialisierung erfolgt dann keine Veränderung mehr.

8.6 Finale Parameter und lokale Variablen [111]

```
public void methode1(final int k) {
    k++; // FEHLER!!
}

public void methode2() {
    final boolean b = true;
    final String s;

    if (b) {
        s = "true";
    }
    else {
        s = "false";
    }
    s.toUpperCase();

    b = false; // FEHLER!!
}
```

Da es ohnehin als ausgesprochen schlechter Stil gilt, Methodenparameter innerhalb der Methode zu verändern, machen viele Programmierer:innen sie direkt bei der Deklaration `final`. Im vorliegenden Beispiel ist der Parameter `k` der Methode `methode1` so deklariert. Der Versuch, die Variable innerhalb der Methode mit `k++` zu ändern führt zu einem Compile-Fehler.

Der Grund für die Regel, dass man Parameter nicht verändern sollte, liegt darin, dass es das Debugging erschwert und den Quelltext schwerer lesbar macht. Wenn man eine Methode betrachtet, die mit einem bestimmten Parameterwert aufgerufen wurde, rechnet man nicht damit, dass dieser Parameter mitten in der Methode plötzlich einen anderen Wert hat.

In `methode2` wird eine normale finale Variable (`b`) und ein blank final `s` verwendet. `s` wird dann, in Abhängigkeit von `b`, mit einem bestimmten Wert initialisiert. Auch hier überprüft der Compiler, ob bei allen Möglichen Pfaden durch das Programm die Variable genau einmal geschrieben wird.

Der Aufruf `s.toUpperCase()` ist unkritisch, weil er den Inhalt des Strings nicht verändert, sondern einen neuen String zurückgibt. Der Versuch, `b` noch einmal zuzuweisen schlägt dann aber mit einem Compiler-Fehler fehl.

8.7 Einkompilieren von Konstanten [112]

Überraschende Optimierung

- Konstanten, d. h. `static final` Variablen eines primitiven Typs, werden direkt in den Verwender einkompiliert, über Klassen und Paketgrenzen hinweg
- Danach gibt es keine Referenz zwischen dem Anbieter der Konstanten und dem Verwender mehr
- Ändert man die Konstante, so muss man zwingend die Verwender neu kompilieren

8.8 Dead Code Elimination [113]

```
public class NoDeadCodeElemination {

    public static boolean SCHALTER = false;

    public static void main(String[] args) {
        if (SCHALTER) {
            System.out.println("Schalter AN");
        }
        else {
            System.out.println("Schalter AUS");
        }
    }
}
```

In diesem Beispiel ist die Variable `SCHALTER` nicht als `static final` deklariert und damit keine Konstante. Betrachtet man den vom Compiler erzeugten Bytecode (mithilfe von `javap`), so sieht man in der folgenden Ausgabe, dass die Bedingung noch vorhanden ist (Zeile 3).

```
public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=1, Args_size=1
   0:   getstatic       #10; //Field schalter:Z
   3:   ifeq    17
   6:   getstatic       #21; //Field java/lang/System.out
   9:   ldc     #27; //String Schalter AN
   11:  invokevirtual   #29; //Method java/io/PrintStream.println
   14:  goto    25
   17:  getstatic       #21; //Field java/lang/System.out
   20:  ldc     #35; //String Schalter AUS
   22:  invokevirtual   #29; //Method java/io/PrintStream.println
```

```
    25:   return
```

Das folgende Beispiel zeigt den Fall, dass die Variable SCHALTER eine echte Konstante ist.

```
public class DeadCodeElemination {

    public static final boolean SCHALTER = false;

    public static void main(String[] args) {
        if (SCHALTER) {
            System.out.println("Schalter AN");
        }
        else {
            System.out.println("Schalter AUS");
        }
    }
}
```



In diesem Fall eliminiert der Compiler den unnötigen und unerreichbaren Code und generiert ein Ergebnis ohne Bedingung. Unerreichbare Code wird auch als **dead code** bezeichnet und der hier vorgestellte Mechanismus daher als **dead code elimination**.

dead code

dead code elimination

```
public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=1, Args_size=1
   0:   getstatic       #20; //Field java/lang/System.out
   3:   ldc     #26; //String Schalter AUS
   5:   invokevirtual   #28; //Method java/io/PrintStream.println
   8:   return
```



8.9 Sealed Classes [117]

Bisher bekannte Möglichkeiten und Beschränkungen von Klassen

Art	Instanzen	Subklassen	überschreiben
normale Klasse	beliebig	ja	ja
Klasse mit privatem Konstruktor	fest	nein	nein
abstrakte Klasse	beliebig	muss	ja
Klasse mit finaler Methode	beliebig	ja	ja, bis auf finale Methode
finale Klasse	beliebig	nein	nein
Enumeration	fest	nein	ja

Enumerationen werden erst später behandelt.

Man sieht an der Auflistung, dass die bisher bekannten Sprachmittel der Entwicklerin einer Klasse nur erlauben, Subklassen ganz zu verbieten oder eben jeder Klassen zu erlauben, Subklassen zu bilden. Dies ist für komplexe Softwareprojekte keine gute Lösung. Deswegen wurde mit Java 17 ein neuer Mechanismus eingeführt, mit dem eine Klasse selbst bestimmen kann, wer von ihr erben darf.

Java 17 führt **Sealed Classes** ein

Sealed Classes

- Syntax: `public sealed class <NAME> permits <CLASS>, <CLASS>, .. {`
- Superklasse kann festlegen, welche Klassen von ihr erben
- Subklassen
  - müssen direkt von der Sealed Class erben
  - müssen im selben Modul liegen
  - müssen zur Compile-Zeit sichtbar sein
- Interfaces können auch versiegelt werden

Sealed Classes wurden zwar bereits mit Java 15 testweise eingeführt (Preview Feature), sind aber erst seit Java 17 stabil und final in den Java-Standard aufgenommen.

Ziele

- Dem Autor einer Klasse oder Schnittstelle Kontrolle darüber zu geben, welcher Code für die Implementierung verantwortlich ist.
- Bereitstellung eines Verfahrens, die Verwendung einer Oberklasse einzuschränken.
- Unterstützung für zukünftige Erweiterungen im Pattern Matching.

Ein Sealed Class in Java beschränkt die Möglichkeiten, wie eine Klasse erweitert werden kann. Eine solche Klasse kann nur von einer festgelegten Liste an Klassen erweitert werden, was sicherstellt, dass nur erwartete Typen verwendet werden. Dies ist nützlich, wenn man sicherstellen möchte, dass nur eine bestimmte Anzahl an Klassen einer bestimmten Hierarchie angehören.

Eine Klasse wird als sealed deklariert und eine Liste mit erlaubten Unterklassen wird angegeben.

Diese Klasse deklariert sich als Sealed Class und gibt an, welche Klassen von ihr erben dürfen: `Circle`, `Square` und `Rectangle`.

Versiegelte Klasse

```
public sealed class Shape
    permits Circle, Square, Rectangle {
}
```

Subklassen einer Sealed Class haben drei Möglichkeiten

- sind `final`: Ende der Vererbungshierarchie
- sind `sealed`: Erlauben bestimmte Subklassen
- sind `non-sealed`: Öffnen sich wieder für beliebige Subklassen

Subklasse: final

```
public final class Circle extends Shape {
    public float radius;
}
```

Subklasse: versiegelt

```
public sealed class Rectangle extends Shape permits FilledRectangle {
    public double length, width;
}
```

Subklasse: nicht versiegelt

```
public non-sealed class Square extends Shape {
   public double side;
}
```

Die Klassen `Circle`, `Square` und `Rectangle` zeigen die drei Möglichkeiten für Subklassen einer Sealed Class:

- `Circle` ist `final`, sodass keine anderen Klassen von `Circle` erben können.
- `Rectangle` ist `sealed` erlaubt ausschließlich der Klasse `FilledRectangle` von ihr zu erben.
- `Square` deklariert sich als `non-sealed`, sodass beliebige weitere Klassen von ihr erben können.

Kapitel 9

Quiz

9.1 Quiz: Überdecken von Attributen [122]

```
class A {
    private int k = 10;
    public int j = 20;

    public void f() {
        System.out.println(k);
    }
}
public class B extends A {
    private int j = 3000;
    public static void main(String[] args) {
        A a = new A();
        A b = new B();
        B b2 = new B();
        System.out.printf("%d %d %d%n", a.j, b.j, b2.j);
        a.f();
    }
}
```

```
20 20 3000
10
```

Das Ergebnis kommt dadurch zu Stande, dass bei überdeckten Attributen die Entscheidung, ob das Attribut vom Sub- oder Supertyp genommen wird, bereits zur Compile-Zeit getroffen wird. Daher geht ausschließlich der statische Typ in diese Entscheidung ein und nicht der dynamische Typ.

In dem vorliegenden Beispiel sind die statischen Typen für die Variablen `a`, `b` und `b2` entsprechend `A`, `A` und `B`. Somit wird beim Lesen des Attributes `j` bei `a.j` und `b.j` auf das Attribut von `A` und bei `b2.j` auf das von `B` zugegriffen.

Beim Aufruf der Methode `f()` über `a.f()` greift die Methode natürlich auf das Attribut der eigenen Klasse zu - woher soll sie auch wissen, dass es ein gleichnamiges Feld irgendwo in der Vererbungshierarchie weiter unter existiert.

9.2 Quiz: Vererbung [124]

```
class A {
}

class B extends A {
}

public class C extends A, B {

    public C() {
        super(null);
    }
}
```

```
de/smits_net/pr2/quiz/vererbung/C.java:6: '{' expected
public class C extends A, B {
                          ^
1 error
```

Da Java nur Einfachvererbung unterstützt, scheitert dieses Beispiel bereits am Compiler, da hier versucht wurde, die Klasse `C` gleichzeitig von `A` und `B` erben zu lassen. Die Java-Syntax erlaubt hinter dem Schlüsselwort `extends` ausschließlich die Angabe einer Klasse. Hier wurden aber zwei angegeben, sodass ein Compile-Fehler entsteht.

9.3 Quiz: Überladen [126]

```
class A {}
class B extends A {}

public class BindingQuiz {
    public static void f(A a) { System.out.print("A "); }
    public static void f(B b) { System.out.print("B "); }

    public static void main(String[] args) {
        A a1 = new A(); A a2 = new B(); B b = new B();
        f(a1);
        f(a2);
        f(b);
    }
```

```
}
```

```
A A B
```

Die Auswahl einer überladenen Methode findet ausschließlich anhand des statischen Typs statt. Es gibt also keinen virtuellen Methodenaufruf für überladene Methoden und damit kein spätes Binden. Stattdessen wählt bereits der Compiler die Methode, die aufgerufen werden soll anhand des statischen Typs aus.

Die statischen Typen der Variablen `a1`, `a2` und `b` sind `A`, `A` und `B`. Somit wird auch zweimal die Methode `f(A)` und nur einmal die Methode `f(B)` gerufen, was zur entsprechenden Ausgabe `A A B` führt.

9.4 Quiz: Überschreiben [128]

```
class A {
    public void f() { System.out.print("A "); }
}

class B extends A {
    public void f() { System.out.print("B "); }
}

public class Polymorphie {
    public static void main(String[] args) {
        A a1 = new A(); A a2 = new B(); B b = new B();
        a1.f(); a2.f(); b.f();
    }
}
```

```
A B B
```

Die Auswahl einer überschriebenen Methode findet anhand des dynamischen Typs statt. Der statische Typ wird vom Compiler nur dazu genutzt, um festzustellen, welche Methoden überhaupt vorhanden sind. Die eigentliche Auswahl der Methode findet aber erst zur Laufzeit statt. Es gibt also einen virtuellen Methodenaufruf damit spätes Binden.

Die dynamischen Typen der Variablen `a1`, `a2` und `b` sind `A`, `B` und `B`. Somit wird einmal die Methode `f()` von `A` und zweimal die Methode `f()` von `B` gerufen, was zur entsprechenden Ausgabe `A B B` führt.

9.5 Quiz: Konstruktoren [130]

```
class A {
    public A() { System.out.println("A"); }
}

class B extends A {
    public B() { System.out.println("B"); }
}

public class ConstructorQuiz {

    public static void main(String[] args) {
        new B();
    }
}
```

```
A
B
```

Java gewährleistet, dass beim Erzeugen einer Instanz einer Subklasse die Konstruktoren aller Superklassen (bis hinauf zu `Object`) durchlaufen werden. Der Grund hierfür ist, dass Attribute, die von Superklassen bereitgestellt werden, korrekt initialisiert werden sollen. Der Konstruktor einer Superklasse läuft grundsätzlich vor dem Konstruktor der Subklasse ab.

Um diese vollständige Initialisierung zu erreichen, wird im vorliegenden Beispiel ein implizites `super()` vom Compiler als erstes Statement in die Konstruktoren von A und B generiert. Hierdurch ruft der Konstruktor von B den Konstruktor von A auf, noch bevor er selbst die Ausgabe machen kann. Als Folge sehen wir als Ausgabe A B.

9.6 Quiz: Statische Methoden [132]

```
public class QuizStatic {

    public static void main(String[] args) {
        String b = null;
        System.out.println(b.valueOf(15));

        System.out.println(((String) null).valueOf(17));

    }
}
```

```
15
17
```

Statische Methoden werden direkt auf der Klasse aufgerufen; ein (gültiges) Objekt wird nicht benötigt. Obwohl man also beim obigen Beispiel eine `NullPointerException` erwarten würde, wird diese nicht geworfen, da vom Compiler nur der statische Typ der Objektreferenz ausgewertet wird, um den Aufruf der Methode auf der entsprechenden Klasse zu generieren. Danach (also insbesondere zur Laufzeit) wird die Objektreferent nicht mehr benötigt und von der Laufzeit gar nicht angefasst.

9.7 Quiz: Static Initializer [134]

```java
public class Initializer {
    static {
        initialize();
    }

    private static boolean initialized = false;

    private static void initialize() {
        initialized = true;
    }

    public static void main(String[] args) {
        System.out.println(initialized);
    }
}
```

```
false
```

Statische Initialisierungen werden strikt von oben nach unten durchgeführt, dazu gehört auch die Initialisierung einer Variable mit einem Wert, wie es mit der Variable `initialized` passiert. Wenn man das Programm von oben nach unten liest, sieht man, dass zuerst die Methode `initialize()` aufgerufen wird, die die Variable auf den Wert `true` setzt, danach aber die Zuweisung erfolgt die Zusammen mit der Deklaration der Variable erfolgt, sodass die Variable wieder auf `false` gesetzt wird.

9.8 Quiz: Initialisierung [136]

```java
public class Ente {
```

```
    private static final Ente INSTANCE = new Ente();
    private static final int NORMAL_GROESSE = Integer.parseInt("12");

    private int groesse = NORMAL_GROESSE - 10;

    public int getGroesse() {
        return groesse;
    }

    public static void main(String[] args) {
        System.out.println(INSTANCE.getGroesse());
    }
}
```

```
-10
```



Die Ausgabe von `-10` mag auf den ersten Blick verwirrend erscheinen. Der Grund liegt darin, dass bei diesem Beispiel zwei Regeln von Java miteinander kollidieren und zwar

- eine Klasse muss vollständig initialisiert werden, bevor Instanzen erzeugt werden und
- statische Initialisierungen werden streng von oben nach unten ausgeführt.

In der Zeile `private static final Ente INSTANCE = new Ente();`

wird eine Ente erzeugt. Hierzu muss aber die Klasse Ente initialisiert werden, was aber die Erzeugung eines Objekts von Ente erfordert. Damit kommt es zu einem Widerspruch der beiden Regeln, der so aufgelöst wird, dass die Initialisierung der Klasse abgebrochen wird, d. h. die Zuweisung von `NORMALE_GROESSE = Integer.parseInt("12")` wird nicht ausgeführt. Hierdurch ist `NORMALE_GROESSE` dann nur mit dem Default-Wert `0` initialisiert, sodass bei der Objekterzeugung `groesse` den Wert `0 - 10` also `-10` hat.

9.9 Quiz: Konstruktor [138]

```
public class NameHolder {
    private String name;
    public void NameHolder() {
        name = "Hallo";
    }
    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        NameHolder s = new NameHolder();
        System.out.println(s);
```



```
    }
}
```

```
null
```


Dieses Beispiel nutzt eine syntaktische Besonderheit von Java aus: Methoden dürfen genauso heißen, wie der Konstruktor, der bekanntermaßen so heißt wie die Klasse (hier also `NameHolder`). Durch das Voranstellen von `void` vor den Konstruktor wird dieser zu einer normalen Methode. Der Compiler generiert dann entsprechend einen Default-Konstruktor. Bei der Erzeugung eines Objektes von `NameHolder` wird hier dann natürlich der generierte Default-Konstruktor aufgerufen, mit der Folge, dass `name` keinen Wert hat, sondern `null` ist.

Index

Abgeleitete Klasse, 7
API, 47
Application Programming Interface, 47

blank final, 50

Cast, 32
Compile-Zeit-Typ, 30
Copy-Konstruktor, 20

dead code, 54
dead code elimination, 54
default, 9
Default-Konstruktor, 17
Destruktoren, 20
dynamischen Typ, 30

Einfachvererbung, 6
Elternklasse, 7

finale Klasse, 49
finale Methode, 49
finale Variable, 49
Finalizer, 20

interning, 22

Kindklasse, 7
Konstruktoren, 16
kovariantem Rückgabetyp, 26

Laufzeit-Typ, 30
liskovsche Substitutionsprinzip, 31

Mehrfachvererbung, 6

Nachbedingung, 34

Oberklasse, 7

package-friendly, 9
polymorph, 28
Polymorphie, 28
Prototyp-Objekt, 20

Sealed Classes, 55
Singletons, 44
Spätes Binden, 33
Static Factory, 45
Statische Attribute, 38
Statische Blöcke, 42
Statische Methoden, 38
statischen Typ, 30
Subklasse, 7
Superklasse, 7

Unterklasse, 7
Unveränderliche Objekte, 21

Vararg-Methoden, 13
Verdecken, 11
verdeckt, 11
Vererbung, 6
virtuellen Methodenaufruf, 30
Vorbedingung, 34

Wither-Methoden, 22

Überladen, 12
überschreiben, 23

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Interfaces und Design



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**

1.1 Fragen [5] . . . 1
1.2 Begriffe [6] . . . 1
1.3 Links zur Java-Insel [7] . . . 2

2 Abstrakte Klassen **3**

2.1 Motivation für abstrakte Klassen [9] . . . 3
2.2 Schritt 1: Methoden hochziehen [12] . . . 4
2.3 Schritt 2: Überschreiben nutzen [15] . . . 6
2.4 Probleme von Schritt 2 [17] . . . 7
2.5 Schritt 3: Implementierung weglassen [19] . . . 7
2.6 Schritt 3: Konsequenzen [21] . . . 8
2.7 Die Lösung [22] . . . 9
2.8 Beispiel: Lösung mit abstrakter Klasse [23] . . . 9
2.9 Abstrakte Klassen [25] . . . 10
2.10 Abstrakte Klassen und Vererbung [26] . . . 11
2.11 Abstrakte Klassen und Konstruktoren [36] . . . 14
2.12 Das Template Method Pattern [38] . . . 15
2.13 Abstrakte Klassen: Zusammenfassung [41] . . . 16

3 Interfaces **18**

3.1 Motivation: Interfaces [43] . . . 18
3.2 Beispiel: Java Interface [44] . . . 18
3.3 Abgrenzung zu abstrakten Klassen [46] . . . 19
3.4 Besonderheiten beim Syntax [47] . . . 20
3.5 Verwendung von Interfaces [48] . . . 20
3.6 Beispiel [49] . . . 21
3.7 Beispiel: Interfaces und Vererbung [51] . . . 22
3.8 Weitere Eigenschaften von Interfaces [56] . . . 24
3.9 Beispiel: Alles auf einmal [57] . . . 25
3.10 Beispiel: Implements-Vererbung [58] . . . 25
3.11 Mehrfachvererbung und Konstanten [61] . . . 26
3.12 Praxis-Tipp: Interfaces und Konstanten [64] . . . 27
3.13 Adapter-Klassen [66] . . . 28
3.14 Comparable [67] . . . 29

3.15 Callbacks [70] 30
3.16 Default-Methoden: Motivation [72] 31
3.17 Lösung: Default-Methoden [74] 32
3.18 Mehrfachvererbung und Default-Methoden [77] 34
3.19 Vererbung von Default-Methoden [78] 34

4 Prinzipien für gutes Design **37**
4.1 SOLID-Prinzipien [82] 37
4.2 STUPID [88] 38

Index **ii**

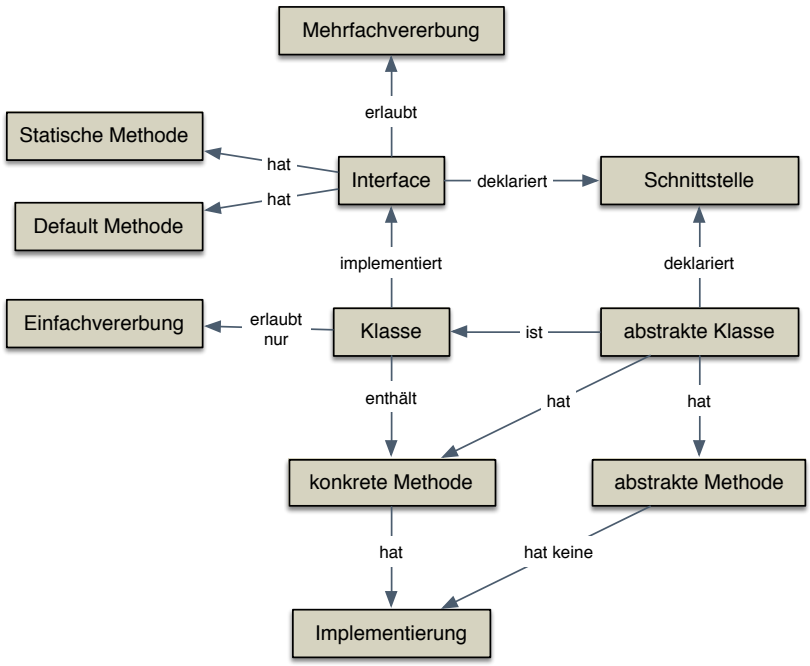
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Was mache ich, wenn ich nur für meine Kinder da bin?
- Wie können Klassen Schnittstellen bereitstellen?
- Wie schreibt man gute Software?
- Was sollte man tunlichst nicht tun?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

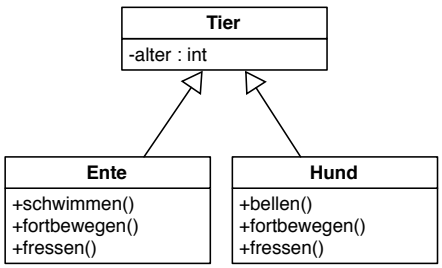
- 7.7 Abstrakte Klassen und abstrakte Methoden
- 8.1 Schnittstellen
- 14.1 SOLIDe Modellierung

Kapitel 2

Abstrakte Klassen

2.1 Motivation für abstrakte Klassen [9]

- Basisklassen vererben ihre Methoden (und Attribute) an ihre Subklassen
- Häufig weiß man, dass alle Subklassen einer Basisklasse eine bestimmte Methode haben werden
- Auf der Ebene der Basisklasse kennt man das Verhalten der Methode jedoch noch nicht
- Erst die Subklassen wissen (können festlegen), wie sich die Methoden genau verhalten sollen
- Man will aber Polymorphie nutzen und die Methoden daher bereits in der Basisklasse definieren




```
public void fuettern(Tier t) {

    if (t instanceof Hund) {
        ((Hund) t).fressen();
    }
    else if (t instanceof Ente) {
        ((Ente) t).fressen();
    }
}
```

Eigentlich können sich alle Tiere fortbewegen und fressen.

```
Hund h = new Hund();
fuettern(h);

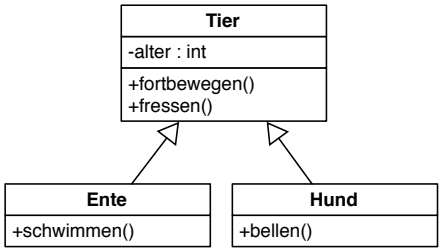
Ente e = new Ente();
fuettern(e);
```

Dieser Code-Abschnitt zeigt eine Methode zum Füttern von Tieren, die den Anspruch hat für alle Arten von Tieren (in diesem Fall also zumindest Hund und Ente) zu funktionieren. Da die Methode `fressen()` aber erst auf der Ebene der konkreten Tiere (Hund und Ente) auftritt, muss in der `fuettern()`-Methode mühselig mit `instanceof` geprüft werden, um welches Tier es sich handelt und mit einem Cast der Zugriff auf die entsprechende Methode hergestellt werden. Ein solches Vorgehen funktioniert zwar, ist aber ausgesprochen schlechter Programmierstil, da für jedes neue Tier, das eingeführt wird (z. B. Ziege), ein neuer else if-Zweig in der `fuettern()`-Methode eingebaut werden muss. Dies hat zur Folge, dass ein grundlegendes Versprechen der objektorientierten Programmierung, nämlich die einfache Erweiterbarkeit durch Einführen neuer Subklassen nicht eingehalten werden kann: für jede neue Subklasse muss die `fuettern()`-Methode erweitert (modifiziert) werden. Vergisst man eine entsprechende Erweiterung in der `fuettern()`-Methode kommt es sogar zu dem Fall, dass ein neu eingeführtes Tier gar nicht gefüttert würde und einfach verhungert.

Wir brauchen also einen Mechanismus, um bereits bei Tier die Methoden festzulegen, die alle Subklassen haben sollten.

2.2 Schritt 1: Methoden hochziehen [12]

Die Methoden können in die Basisklasse verlegt werden




```
public class Tier {
    private int alter;
    public void fortbewegen() { /* Tiere bewegen sich, aber wie? */ }
    public void fressen() { /* Tiere fressen, aber wie? */ }
}

public class Hund extends Tier {
    public void bellen() {
        // Maul auf und los
    }
}
```

```
public class Ente extends Tier {

    public void schwimmen() {
        // mit den Füßen paddeln
    }
}
```

```
public void fuettern(Tier t) {
    // Futter bereitstellen
    t.fressen();
}
```



```
Hund h = new Hund();
fuettern(h);

Ente e = new Ente();
fuettern(e);
```



- Erst Hund und Ente, wissen wie die Methoden konkret aussehen sollen
- Alle Tiere können sich fortbewegen, aber wie genau?
- Alle Tiere können fressen, aber wie genau?

Ein erster Ansatz, um das beschriebene Problem zu lösen, wäre es die `fressen()`- und `fortbewegen()`-Methoden in die Basisklasse `Tier` zu ziehen und so dafür zu sorgen, dass sie an alle Subklassen (alte wie neue) vererbt wird. Die `fuettern()`-Methode kann dann ganz auf `instanceof` verzichten und sicher davon ausgehen, dass alle Tiere eine entsprechende `fressen()`-Methode haben.

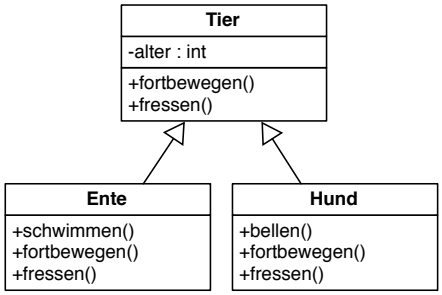
Hier ergibt sich aber das Problem, dass wir auf der Ebene der Tier-Klasse nicht festlegen können, wie die einzelnen Tiere genau fressen.

- Obwohl Tier eine Abstraktion von konkreten Tieren darstellt (in unserem Modell gibt es zwar Enten und Hunde aber keine Tiere), kann man weiterhin Instanzen von Tier erstellen.
- Die Klasse Tier kann nicht wissen, wie sich Enten und Hunde fortbewegen und wie sie fressen. Man kann diese Methoden zwar auf allen Tieren aufrufen, das Verhalten passt aber nicht zum jeweiligen Tier, weil es von der Klasse Tier stammt.

Wir brauchen also eine Möglichkeit, für Ente und Hund (und alle weiteren Subklassen) das Verhalten der Methoden anzupassen.

2.3 Schritt 2: Überschreiben nutzen [15]

- Subklassen überschreiben die Methoden der Superklasse
- Die Implementierung in der Superklasse (hier Tier) ist damit bedeutungslos, die Schnittstelle aber festgelegt



Das Problem, dass Tier das konkrete Verhalten von Enten und Hunden beim Fressen und Fortbewegen nicht kennen kann, können wir dadurch lösen, dass wir die Methoden in allen Subklassen überschreiben und mit dem gewünschten Verhalten der jeweiligen Tiere ersetzen. Durch den virtuellen Methodenaufruf wird gewährleistet, dass Verwender von Enten und Hunden bei einer Referenz vom Typ `Tier` die richtigen Methoden der konkreten Tiere aufrufen.

```
public class Hund extends Tier {
    public void bellen() { /* Maul auf und los */ }

    @Override
    public void fortbewegen() { /* auf allen Vieren dackeln */ }

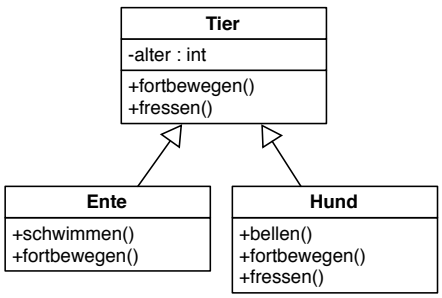
    @Override
    public void fressen() { /* Maul auf und runter mit Futter */ }
}

public class Ente extends Tier {
    public void schwimmen() { /* mit den Füßen paddeln */ }

    @Override
    public void fortbewegen() { /* watscheln */ }

    @Override
    public void fressen() { /* Tauchen und Schnäbeln */ }
}
```

2.4 Probleme von Schritt 2 [17]



- Es können Instanzen der Superklasse erzeugt werden, obwohl sie keine sinnvolle Implementierung der Methoden hat
- Subklassen können vergessen, die Methoden zu überschreiben

```
Hund h = new Hund();
h.fortbewegen();

// Es wird eine Instanz von Tier erstellt, obwohl es in der realen
// Welt keine Tiere, sondern nur Hunde und Enten gibt.
Tier t = new Tier();


Tier e = new Ente();
// Ente hat leider vergessen die fressen()-Methode zu überschrieben.
e.fressen();
```

Bei der Variante mit dem Überschreiben bleiben jedoch zwei grundlegende Probleme ungelöst:

- Es ist weiterhin möglich Objekte von Tier anzulegen, obwohl dies (wie weiter oben beschrieben) sinnlos ist und
- es kann vorkommen, dass Subklassen vergessen eine der beiden Methoden zu überschreiben, sodass doch noch die (leere) Implementierung von Tier zum Tragen kommt.

Uns fehlt also ein Sprachkonstrukt, mit dem wir Subklassen dazu zwingen können ausgewählte Methoden der Superklasse zu überschreiben und mit dem wir klarmachen können, dass es keine Objekte vom Typ der Superklasse geben soll.

2.5 Schritt 3: Implementierung weglassen [19]

Wenn die Implementierungen der Methoden der Superklasse sowieso überschrieben werden, warum lässt man sie dann nicht ganz weg?

```
public class Tier {
```

```
    private int alter;

    public void fortbewegen() {
        // mhmm... Tiere bewegen sich, aber wie?
    }

    public void fressen() {
        // mhmm... Tiere fressen, aber wie?
    }
}
```

Wenn die Implementierungen der Methoden der Superklasse sowieso überschrieben werden, warum lässt man sie dann nicht ganz weg?



```
public abstract class Tier {

    private int alter;

    public abstract void fortbewegen();

    public abstract void fressen();
}
```

Wenn wir ohnehin alle Subklassen zwingen wollen, ausgewählte Methoden der Superklasse zu überschreiben, dann ist die konkrete Implementierung dieser Methoden egal, da sie ohnehin immer überschrieben werden. Es ist also nur konsequent, diese Implementierungen dann ganz wegzulassen.

abstrakten Methoden

Mithilfe von **abstrakten Methoden** (abstract methods) hat man die Möglichkeit, Methoden zu definieren, ohne eine Implementierung anzugeben. Zu erkennen ist das am fehlenden Methodenrumpf ({ ... }), der durch ein Semikolon ersetzt wird. Weiterhin erfordert die Java-Syntax, dass man die Methoden mit dem Schlüsselwort `abstract` kennzeichnet, obwohl eigentlich der fehlende Methodenrumpf ausgereicht hätte. Man hat sich aber für eine explizite Kennzeichnung entschieden, um ein versehentliches Vergessen der Implementierung erkennen zu können.

abstract

2.6 Schritt 3: Konsequenzen [21]

- Methode ohne Implementierung = **abstrakte Methode** (abstract method) — abstrakte Methode
- Klasse mit abstrakten Methoden = **abstrakte Klasse** (abstract class) — abstrakte Klasse
- abstrakte Methoden haben *keine Implementierung*
  ⇒ man kann sie nicht aufrufen, bilden nur eine Schnittstelle
- Subklassen *müssen* die abstrakten Methoden *implementieren*
  ⇒ man kann das „überschreiben“ nicht mehr vergessen

- man muss verhindern, dass abstrakte Methoden auf einer Instanz der abstrakten Klasse aufgerufen werden (hier Tier)
  ⇒ man verbietet, Objekte abstrakter Klassen zu erzeugen

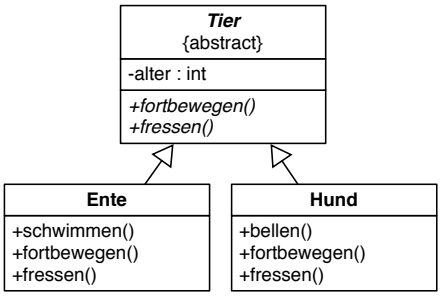
Durch die Einführung abstrakter Klassen löst man sehr elegant das beschriebene Problem: Man kann die Methoden bereits auf Ebene der Superklasse deklarieren, gibt aber keine Implementierung an. Hierdurch wird das Vorhandensein der Methoden garantiert, das konkrete Verhalten festzulegen wird aber den Subklassen überlassen.

Sobald eine Klasse abstrakte Methoden hat, wird sie selbst zur abstrakten Klasse. Die Konsequenz ist, dass man keine Objekte (Instanzen) der abstrakten Klasse mehr erzeugen kann. So wird sicher verhindert, dass die Methoden ohne Implementierung aufgerufen werden, denn ohne Objekt, kann man keine Methoden auf diesem aufrufen.

Dadurch, dass man die Subklassen (über den Compiler) dazu zwingt, die abstrakten Methoden zu implementieren, kann man sicher sein, dass eine Implementierung vorhanden ist.

2.7 Die Lösung [22]

- Die Methoden von Tier werden abstrakt gemacht
- Die Subklassen müssen die Methoden implementieren



Tier definiert abstrakte Methoden, die von Ente und Hund implementiert werden müssen.

2.8 Beispiel: Lösung mit abstrakter Klasse [23]

```
public abstract class Tier {
    private int alter;

    public abstract void fortbewegen();
    public abstract void fressen();
}


public class Hund extends Tier {

    public void bellen() { /* Maul auf und los */ }
```

```
    @Override
    public void fortbewegen() { /* auf allen Vieren dackeln */ }

    @Override
    public void fressen() { /* Maul auf und runter mit dem Futter */ }
}
```

```
public class Ente extends Tier {

    public void schwimmen() { /* mit den Füßen paddeln */ }

    @Override
    public void fortbewegen() { /* watscheln */ }

    @Override
    public void fressen() { /* abtauchen und Schnabel auf */ }
}

public class Verwender {

     public void fuettern(Tier t) {
        // Futter bereitstellen
        t.fressen();
    }
}
```

2.9 Abstrakte Klassen [25]

- *Abstrakte Methoden*
  - deklarieren nur eine Schnittstelle
  - haben keinen Methodenrumpf
  - werden durch das Schlüsselwort `abstract` gekennzeichnet
  - verhalten sich bei der Polymorphie wie normale Methoden
- *Abstrakte Klassen*
  - haben abstrakte Methoden
  - sind selbst abstrakt (Schlüsselwort `abstract`)
  - können als Typ für Variablen dienen (Polymorphie)
  - erlauben keine Objekterzeugung

Man kann in Java abstrakte Methoden deklarieren, die nur eine Schnittstelle aber keine Implementierung definieren, d. h. sie enthalten keinen Methodenrumpf, z. B. `public abstract int berechneZahlung();`

Das Schlüsselwort `abstract` muss in der Methodendeklaration angegeben werden; nur dann darf man den Methodenrumpf weglassen. Das Gegenteil einer abstrakten Methode (abstract method) ist übrigens die **konkrete Methode** **(concrete Method)**.

konkrete Methode

Eine Klasse, in der mindestens eine abstrakte Methode vorkommt, ist selbst wieder abstrakt und muss daher mit dem Schlüsselwort `abstract` gekennzeichnet werden:
`public abstract class Mitarbeiter`

Im Prinzip ist dieses zusätzliche `abstract` in der Klassendefinition unnötig, da es der Compiler an den abstrakten Methoden erkennen könnte, trotzdem muss man es noch einmal hinschreiben.

Zu beachten ist, dass man abstrakte Klassen deklarieren kann, die *keine* abstrakten Methoden enthalten. Von diesen Klassen kann man dann keine Objekte erzeugen, es werden aber keine abstrakten Methoden an Subklassen weitergegeben.

```
public abstract class AbstrakteKlasse {
    public int m() {
        return 1;
    }
}
```

Von abstrakten Klassen kann man keine Objekte erzeugen, was keinen Sinn ergeben würden, da für mindestens eine Methode keine Implementierung existieren würde. Natürlich können abstrakte Klassen neben den abstrakten Methoden normale Methoden enthalten.

Der gesamte Satz von Methoden (abstrakte und konkrete) wird weitervererbt. Hierdurch kann ein Verwender einer Referenz vom Typ einer abstrakten Klasse alle Methoden aufrufen.

2.10 Abstrakte Klassen und Vererbung [26]

Subklassen abstrakter Klassen

- implementieren alle abstrakten Methoden *oder*
- implementieren nur einen Teil der abstrakten Methoden, müssen dann aber selbst wieder abstrakt sein

Abstrakte Methoden

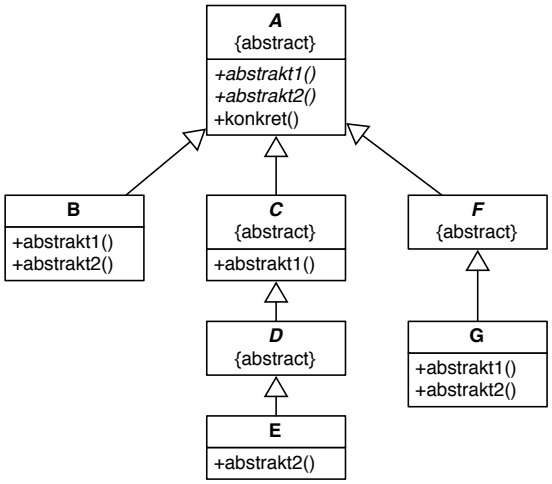
- werden (einschließlich der Abstraktheit) weitervererbt
- können über mehrere Vererbungsstufen hinweg schrittweise implementiert werden
- können nur `public` oder `protected` sein

Die hier vorgestellten Regeln werden im Folgenden anhand eines konkreten Beispiels erläutert.

Würde man für abstrakte Methoden andere Sichtbarkeiten als `public` oder `protected` zulassen, könnte man abstrakte Klassen konstruieren bei denen die Subklassen gar nicht mehr in der Lage wären, eine Implementierung zur Verfügung zu stellen. Da dies offensichtlich Unsinn ist, hat man die Sichtbarkeit auf `public` oder `protected` eingeschränkt. Diese ist konsistent

mit der Maßgabe, dass abstrakte Methoden den **Vertrag** (contract) einer Klasse nach außen darstellen. Ein Vertrag, den die Subklasse nicht sehen könnte, ist sinnlos.

Vertrag



Das Beispiel zeigt deutlich, wie die Vererbung von abstrakten Methoden funktioniert. Die abstrakte Klasse `A` enthält zwei abstrakte Methoden (`abstrakt1()` und `abstrakt2()`) und eine konkrete Methode. Im Folgenden wird nur die Vererbung der abstrakten Methoden weiter beschrieben. Dass die konkreten Methoden vererbt werden, ist klar.

- Die Klasse `B` implementiert beide gerbten abstrakten Methoden und ist damit eine konkrete Klasse.
- Die Klasse `C` implementiert nur eine abstrakte Methode, nämlich `abstrakt1()`. Somit fehlt die Implementierung von `abstrakt2()` wodurch `C` selbst wieder abstrakt ist, da nicht alle geerbten abstrakten Methoden implementiert wurden.
- Die Klasse `D` erbt von `C` die abstrakte Methode `abstrakt2()`. Da sie diese nicht implementiert, ist sie selbst wieder abstrakt.
- Die Klasse `E` erbt von `D` die abstrakte Methode `abstrakt2()` und implementiert sie, wodurch sie zu einer konkreten Klasse wird.
- Die Klasse `F` implementiert keine einzige der beiden ererbten abstrakten Methoden und ist dadurch selbst abstrakt.
- Die Klasse `G` erbt von `F` zwei abstrakte Methoden, implementiert sie aber beide und ist dadurch eine konkrete Klasse.

```
public abstract class A {

    public abstract void abstrakt1();

    public abstract void abstrakt2();

    public void konkret() {
        System.out.println("Konkret Alter!");
```

```
    }
}
```

```
public class B extends A {

    public void abstrakt1() {
        System.out.println("Abstrakt 1 aus B");
    }

    public void abstrakt2() {
        System.out.println("Abstrakt 2 aus B");
    }
}

A b = new B();
b.konkret();  b.abstrakt1();  b.abstrakt2();
```



```
Konkret Alter!
Abstrakt 1 aus B
Abstrakt 2 aus B
```



```
public abstract class C extends A {
    public void abstrakt1() {
        System.out.println("Abstrakt 1 aus C");
    }
}
public abstract class D extends C { /* empty */ }

public class E extends D {
    public void abstrakt2() {
        System.out.println("Abstrakt 2 aus E");
    }
}
A e = new E(); e.konkret(); e.abstrakt1(); e.abstrakt2();
```



```
Konkret Alter!
Abstrakt 1 aus C
Abstrakt 2 aus E
```



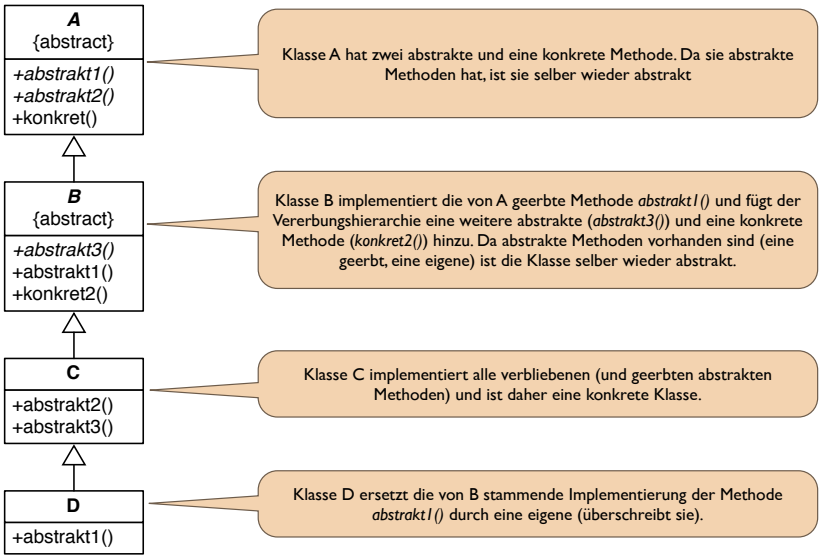
```
public abstract class F extends A { /* empty */ }

public class G extends F {
```

```
    public void abstrakt1() {
        System.out.println("Abstrakt 1 aus G");
    }

    public void abstrakt2() {
        System.out.println("Abstrakt 2 aus G");
    }
}
A g = new G(); g.konkret(); g.abstrakt1(); g.abstrakt2();
```

```
Konkret Alter!
Abstrakt 1 aus G
Abstrakt 2 aus G
```



2.11 Abstrakte Klassen und Konstruktoren [36]

- Auch abstrakte Klassen brauchen Konstruktoren – obwohl man von ihnen keine Objekte erzeugen kann
- Konstruktoren abstrakter Klassen dienen dazu, die Attribute der Klasse zu initialisieren
- Es gelten die allgemeinen Regeln für Konstruktoren
  - Ist kein Konstruktor deklariert, wird vom Compiler ein Default-Konstruktor generiert
  - In der ersten Zeile des Konstruktors steht entweder ein `this()`- oder ein `super()`-Aufruf

- Wird kein `this()` oder `super()` angegeben, generiert der Compiler ein implizites `super()`
- In mindestens einem Konstruktor steht ein `super()`-Aufruf

```java
public abstract class Mitarbeiter {

    protected String name;
    protected double gehalt;
    protected Date geboren;

    public Mitarbeiter(String name, double gehalt, Date geboren) {
        this.name = name;
        this.gehalt = gehalt;
        this.geboren = geboren;
    }

    public Date getGeburtsdatum() {
        return geboren;
    }

    protected abstract int berechneZahlung();
}
```

2.12 Das Template Method Pattern [38]

Template Method Pattern

Das **Template Method Pattern** ist ein Entwurfsmuster, das dazu verwendet wird, die Struktur einer Algorithmus-Implementierung festzulegen, während die Schritte des Algorithmus selbst von Unterklassen definiert werden. Es ermöglicht, dass eine abstrakte Superklasse einen allgemeinen Algorithmus definiert, der aus einer Reihe von Schritten besteht, von denen einige von Unterklassen implementiert werden müssen.

Template Method

Im Template Method Pattern hat die abstrakte Superklasse eine Methode, die als **Template Method** bezeichnet wird, die die Schritte des Algorithmus in der richtigen Reihenfolge aufruft. Einige dieser Schritte sind in der abstrakten Superklasse definiert und können von Unterklassen genutzt werden, während andere Schritte als abstrakte Methoden in der Superklasse definiert sind und von Unterklassen implementiert werden müssen.

```java
public abstract class Mitarbeiter {

    public final double bonus() {
        return zielbonus() * zielerreichung();
    }

    public abstract double zielerreichung();
    public abstract double zielbonus();
}
```

```java
public class Angestellter extends Mitarbeiter {

    private double zielbonus = 1000.0;
    private double zielerreichung = 0.98;

    public double zielbonus() {
        return zielbonus;
    }

    public double zielerreichung() {
        return zielerreichung;
    }
}
```

Innerhalb der abstrakten Klasse können die abstrakten Methoden von konkreten Methoden genutzt werden, sodass man über das **Template Method-Pattern** eine bestimmte Implementierung eines Algorithmus vorgeben kann, in den sich die Subklassen durch Implementierung der abstrakten Methoden einhängen können.

Template Method-Pattern

```java
public class Vorstand extends Mitarbeiter {

    private double firmengewinn = 300000.0;

    public double zielbonus() {
        return 0.5 * firmengewinn;
    }

    public double zielerreichung() {
        return 1.00;
    }
}
```

2.13 Abstrakte Klassen: Zusammenfassung [41]

Abstrakte Klassen

- Fördern die *Trennung von Schnittstelle und Implementierung*
- Legen als Basisklassen Eigenschaften von Subklassen fest, ohne die Implementierung vorgeben zu müssen
- Erlauben durch Polymorphie und virtuellen Methodenaufruf elegante objektorientierte Konstruktionen
- Sind *echte Superklassen*: es besteht eine ist-eine (is-a) Beziehung zwischen Super- und Subklasse

Eine abstrakte Klasse ist eine spezielle Art von Klasse in der Programmierung, die nicht direkt instanziiert werden kann. Sie dient als Vorlage für andere Klassen, die von ihr abgeleitet werden. Eine solche Klasse enthält in der Regel abstrakte Methoden, die von den abgeleiteten Klassen implementiert werden müssen. Sie kann auch konkrete Methoden und Felder enthalten, die von den abgeleiteten Klassen genutzt werden können.

Kapitel 3

Interfaces

3.1 Motivation: Interfaces [43]

Wozu Interfaces?

öffentliche Schnittstelle

- Eine **öffentliche Schnittstelle** (public interface) stellt einen Vertrag zwischen einer Klasse und ihren Verwendern dar
- Eine Klasse kann mehrere, verschiedene Verträge eingehen
- Nicht immer stammt die Schnittstelle von den Eltern
- Manche Verträge bestehen außerhalb der Vererbungshierarchie
- Compiler soll sicherstellen, dass der Vertrag eingehalten wird

Interface

Ein **Interface** in Java erlaubt, einen öffentlichen Vertrag außerhalb der Vererbungsbeziehungen zu definieren und zu implementieren

interface

Ein Interfaces wird, in Abgrenzung zu einer Klasse, durch das Schlüsselwort `interface` gekennzeichnet.

3.2 Beispiel: Java Interface [44]

```
public interface Flieger {
    public void starten();
    public void landen();
    public void fliegen();
}

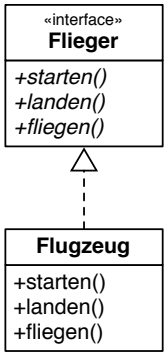
public class Flugzeug implements Flieger {

    public void fliegen() { // ...
    }

    public void landen() { // ...
    }

    public void starten() { // ..
```

```
    }
}
```



3.3 Abgrenzung zu abstrakten Klassen [46]

Java-Interfaces sind ähnlich zu abstrakten Basisklassen, mit folgenden Unterschieden

- *Alle* Methoden sind abstrakt
- *Keine* Attribute
- *Keine* Konstruktoren
- **Mehrfachvererbung** ist möglich

Mehrfachvererbung

- Interfaces stehen neben der Vererbungshierarchie
- Von einem Interface erbt man nicht, man *implementiert* es
- Eine Klasse kann beliebig viele Interfaces implementieren

Interfaces sind in ihrem Zweck ähnlich abstrakten Basisklassen: sie definieren einen Vertrag, den die Klassen, die das Interface implementieren, einhalten müssen. Trotzdem gibt es einige Unterschiede zwischen abstrakten Basisklassen und Interfaces:

- Ein Interface kann *keine Implementierung* enthalten, d. h. alle Methoden eines Interfaces sind immer abstrakt und daher ohne Rumpf. Mit Java 8 gibt es allerdings eine Ausnahme zu dieser Regel, die weiter unten beschrieben wird.
- Ein Interface kann *keine Daten* tragen, d. h. es enthält keiner Attribute. Durch das Implementieren eines Interfaces werden also der implementierenden Klasse keine neuen Felder hinzugefügt. Einzige Ausnahme von dieser Regel ist, dass Interfaces Konstanten enthalten dürfen, also static-final-Felder, die später noch genauer betrachtet werden.
- Interfaces besitzen *keine Konstruktoren*, da man von ihnen keine Objekte erzeugen kann und sie neben der Vererbungshierarchie stehen.
- Zwischen Interfaces kann es (im Gegensatz zu Klassen) Mehrfachvererbung geben. D. h. ein Interface kann selbst von mehreren Interfaces abgeleitet sein. Dies ist insofern problemlos, als bei Interfaces weder Membervariablen noch Implementierungen

zugelassen sind. Hierdurch werden die Hauptproblemquellen von Mehrfachvererbung ausgeschlossen.

- Interfaces stehen neben der Vererbungshierarchie: neben der Hierarchie der Klassen gibt es eine weitere Vererbungshierarchie zwischen den Interfaces. Klassen implementieren daher Interfaces und erben nicht von ihnen. Die Klassen können dann beliebige (und beliebig viele) Interfaces implementieren.
- Es gibt keine Beschränkung, wie viele Interfaces eine Klasse implementieren kann.

3.4 Besonderheiten beim Syntax [47]

- Methoden eines Interfaces sind immer `public abstract`
- Konstanten sind immer `public static final`

```
public interface Flieger {
    int CONSTANT = 12;
    void starten();
    void landen();
    void fliegen();
}
```

```
public interface Flieger {
    public static final int CONSTANT = 12;
    public abstract void starten();
    public abstract void landen();
    public abstract void fliegen();
}
```

Da ein Interface einen öffentlichen Vertrag definiert, sind alle Methoden des Interfaces immer `public` und da es keine Implementierung enthalten darf sind sie immer auch abstrakt. Aus diesem Grund kann man das `public abstract` an den Methoden auch weglassen, der Compiler fügt es automatisch hinzu.

Da ein Interface keine Membervariablen enthalten kann, werden alle Variablendeklarationen vom Compiler explizit mit einem `public static final` versehen.

3.5 Verwendung von Interfaces [48]

Interfaces sind genauso Typen wie Klassen

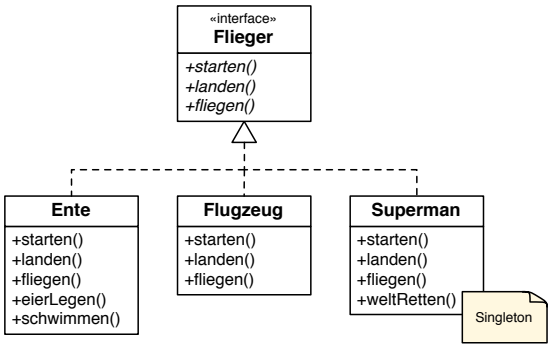
- Referenzvariablen können auch vom Typ eines Interfaces sein
- Objekte einer Klasse, die ein Interface implementiert, können über eine Variable vom Interface-Typ adressiert werden (Polymorphie)
- Über den Interface-Typ sind nur die Methoden des Interfaces sichtbar

```
public class Flughafen {

    public static void main(String[] args) {
        Flieger flieger = new Flugzeug();
        flieger.landen();
    }
}
```

Interfaces verhalten sich genauso wie abstrakte Basisklassen. Objekte einer Klasse, die ein Interface implementiert, können über Variablen vom Typ des Interfaces referenziert werden. Die Polymorphie bezieht sich also nicht nur auf die Vererbungshierarchie, sondern auf die Implementierungsbeziehungen.

3.6 Beispiel [49]



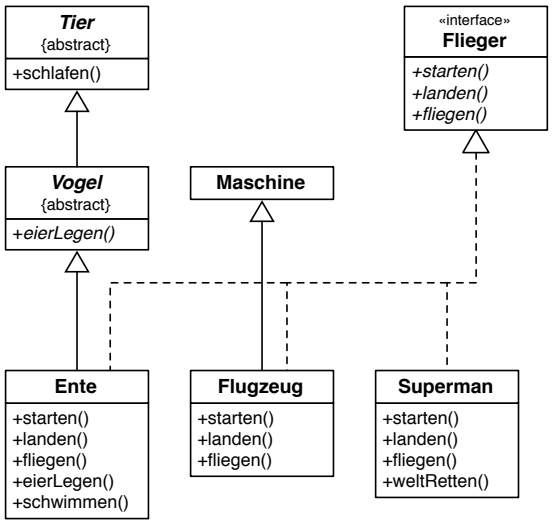
Das Diagramm zeigt ein Interface und drei Klassen, die das Interface implementieren. Zwischen den Klassen besteht keine Gemeinsamkeit: Eine Ente ist ein Tier, ein Flugzeug ist eine Maschine und Superman ist ein Außerirdischer, der vom Planeten Krypton auf die Erde gekommen ist, als sein Planet zerstört wurde. Da Superman der einzige Überlebende von Krypton ist, handelt es sich bei ihm um ein Singleton – eine Klasse, von der es nur ein Objekt geben darf. (Details zu Singletons folgen später.)

Trotzdem haben alle drei Klassen eine Gemeinsamkeit, nämlich, dass sie die Fähigkeit besitzen, zu fliegen. Dies drücken sie dadurch aus, dass sie gemeinsam ein Interface `Flieger` implementieren.

```
public interface Flieger {
    void starten();
    void landen();
    void fliegen();
}
```

```
public class Superman implements Flieger {
    public void fliegen() { ... }
    public void landen() { ... }
    public void starten() { ... }
    public void weltRetten() { ... }
}
```

3.7 Beispiel: Interfaces und Vererbung [51]



Die Klassen können neben der Implementierungsbeziehung zum Interface von anderen Klassen erben. Dass Interface darf aber erst auf der Ebene der Vererbungshierarchie implementiert werden, ab der klar ist, dass alle Subklassen sicher die Fähigkeit zu fliegen besitzen. Daher implementiert erst `Ente` das Interface, da es Vögel gibt, die nicht fliegen können.

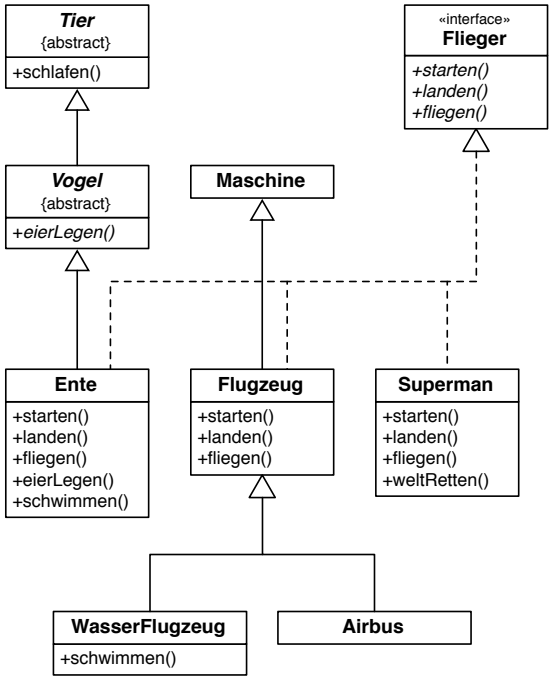
```
public abstract class Tier {
    public void schlafen() { ... }
}

public abstract class Vogel extends Tier {
    public abstract void eierLegen();
}

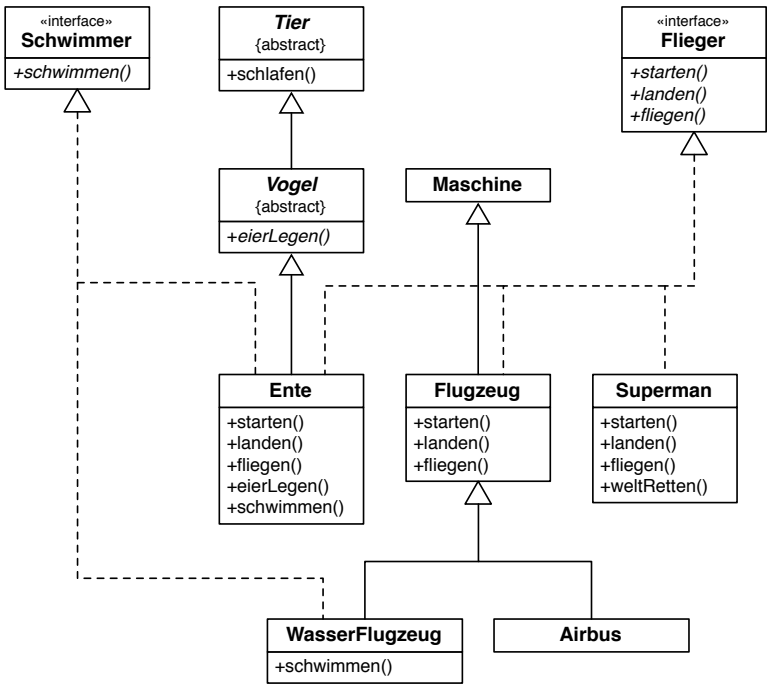
public class Ente extends Vogel implements Flieger {

    public void eierLegen() { ... }
    public void schwimmen() { ... }
    public void fliegen() { ... }
    public void landen() { ... }
    public void starten() { ... }
```

```
}
```



Auch nach der Implementierung des Interfaces können noch weitere Klassen abgeleitet werden. Die Tatsache, dass das Interface von der Elternklasse (hier `Flugzeug`) implementiert wurde, wird an alle Subklassen vererbt. Also implementieren hier auch die Klassen `WasserFlugzeug` und `Airbus` das Interface `Flieger`.



Weitere Interfaces können eingeführt werden, hier z. B. `Schwimmer`, das von allen Klassen implementiert wird, die schwimmen können.

```
public class Ente extends Vogel
    implements Schwimmer, Flieger {

    public void eierLegen() { ... }
    public void schwimmen() { ... }
    public void fliegen() { ... }
    public void landen() { ... }
    public void starten() { ... }
}

public class Wasserflugzeug extends
     Flugzeug implements Schwimmer {

    public void schwimmen() { ... }
}
```

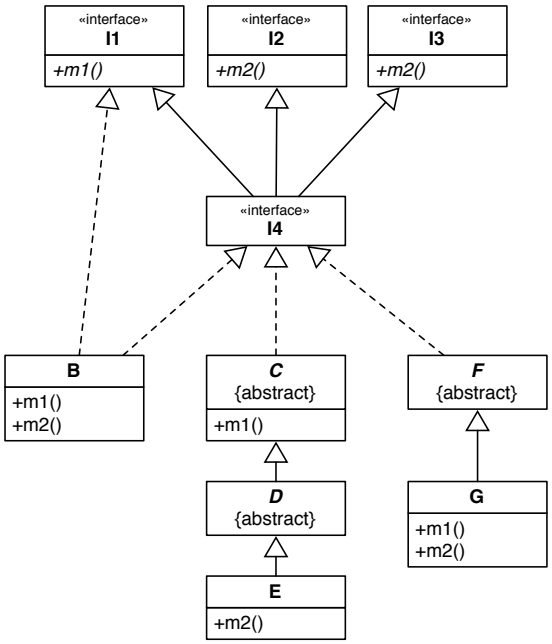
3.8 Weitere Eigenschaften von Interfaces [56]

- Man kann mit `instanceof` auf Interfaces prüfen
- Die `implements`-Beziehung wird weitervererbt
- Klassen, die ein Interface implementieren,
  - implementieren alle Methoden des Interfaces *oder*

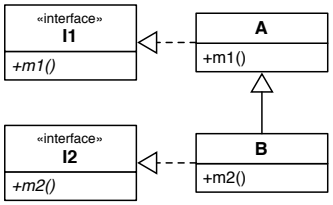
- implementieren nur einen Teil der Methoden, werden dann aber selbst abstrakt

Über die Regel, dass ein Interface entweder vollständig implementiert wird oder die Klasse wird abstrakt, wird eine Verbindung zwischen Interfaces und abstrakten Klassen hergestellt.

3.9 Beispiel: Alles auf einmal [57]



3.10 Beispiel: Implements-Vererbung [58]




```
A a = new A();
B b = new B();
System.out.println(a instanceof I1); // --> true
System.out.println(a instanceof I2); // --> false
System.out.println(b instanceof A); // --> true
System.out.println(b instanceof I1); // --> true
System.out.println(b instanceof I2); // --> true
```

```java
public interface I1 {
    public abstract void m1();
}

public interface I2 {
    public abstract void m2();
}

public class A implements I1 {
    public void m1() { /* leer */ }
}

public class B extends A implements I2 {
    public void m2() { /* leer */ }
}
```

```java
A a = new A();
B b = new B();
System.out.println(a instanceof I1); // --> true
System.out.println(a instanceof I2); // --> false
System.out.println(b instanceof A); // --> true
System.out.println(b instanceof I1); // --> true
System.out.println(b instanceof I2); // --> true
```



3.11 Mehrfachvererbung und Konstanten [61]

```java
public interface I1 {
    public static final int KONSTANTE = 42;
}
public interface I2 {
    public static final int KONSTANTE = 23;
}
public class A implements I1, I2 {
    public static void main(String[] args) {
        System.out.println(KONSTANTE); // FEHLER!!
    }
}
```



```
pr2/mehrfachvererbung/A.java:9: reference to KONSTANTE is
ambiguous, both variable KONSTANTE in pr2.mehrfachvererbung.I1
and variable KONSTANTE in pr2.mehrfachvererbung.I2 match
        System.out.println(KONSTANTE);
```



Der Compiler kann in der Klasse A nicht wissen, welche Variable mit dem Namen KONSTANTE gemeint ist. Daher kommt es wegen der Mehrdeutigkeit zu einem Compiler-Fehler. Dieses Problem kann man leicht auflösen, indem man explizit angibt, welche Konstante man meint.

```
public class A implements I1, I2 {

    public static void main(String[] args) {
        System.out.println(I1.KONSTANTE);
        System.out.println(I2.KONSTANTE);
    }
}
```

3.12 Praxis-Tipp: Interfaces und Konstanten [64]

Interfaces sollten nicht zur Definition von Konstanten verwendet werden

So nicht…

```
public interface Konstanten {
    public static final double PI = 3.14159265;
}
```

```
public class Mathe implements Konstanten {

    public double kreisFlaeche(double radius) {
        return PI*radius*radius;
    }
}
```

sondern so…

```
public final class Konstanten {

    private Konstanten() {}

    public static final double PI = 3.14159265;
}
```

```
import static pr2.Konstanten.PI;

public class Mathe {

    public double kreisFlaeche(double radius) {
```

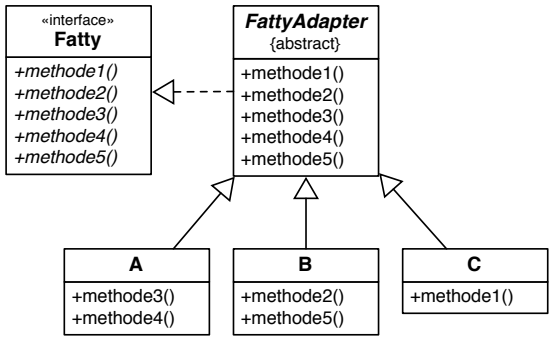
```
        return PI*radius*radius;
    }
}
```

Ein früher häufig propagiertes Muster war die Verwendung von Interfaces zur Definition von Konstanten. Da man die Konstanten dann mit einem einfachen `implements` in der Klasse bekannt machen konnte, konnte man die Angabe der Klasse, aus der man die Konstanten verwenden wollte, vermeiden. Dieses Vorgehen ist aber spätestens seit der Einführung von `static`-Imports ohne Vorteil, hat aber signifikante Nachteile, da es einen weiteren Typ einführt und suggeriert, dass der Implementierende des Interfaces einen besonderen Kontrakt einhält.

Da es sich hier um einen Missbrauch von Interfaces handelt und man dasselbe Ergebnis auf einem anderen Weg erreichen kann, ist von der Verwendung von Interfaces für das Einführen von Konstanten dringend abzuraten.

3.13 Adapter-Klassen [66]

Adapter stellen eine Basisimplementierung eines Interfaces zur Verfügung



Gerade im Bereich der grafischen Oberflächen muss man häufig Interfaces implementieren, um auf Ereignisse zu reagieren (z. B. dass ein Knopf gedrückt wurde). Die Interfaces haben viele Methoden, man braucht aber oft nur einen Bruchteil davon. Dies hat zur Folge, dass man viele Methoden leer implementiert, weil man alle Methoden des Interfaces implementieren muss.

Eine Lösung für dieses Problem ist die Verwendung von Adapter-Klassen. Eine solche Klasse implementiert das Interface vollständig, wenn auch leer, d. h. die Methoden machen nichts. Anstatt das Interface zu implementieren, erbt man dann von der Adapter-Klasse und überschreibt nur noch die relevanten Methoden.

Der Nachteil einer Adapter-Klasse ist, dass man bei ihrer Verwendung, wegen der Einfachvererbung, nicht mehr von einer anderen Klasse erben kann.

3.14 Comparable [67]

```
public class Telefonnummer implements Comparable {
    private int vorwahl; private int nummer;
    public int compareTo(Object o) {
        Telefonnummer n = (Telefonnummer) o;
        if (vorwahl > n.vorwahl) {
            return 1;
        } else if (vorwahl < n.vorwahl) {
            return -1;
        } else { // vorwahl == n.vorwahl
            if (nummer > n.nummer) {
                return 1;
            } else if (nummer < n.nummer) {
                return -1;
            }
        }
        return 0;
    }
}
```

Die Java-Klassenbibliothek enthält bereits eine ganze Reihe von Interfaces, z. B. `Comparable`. `Comparable` erlaubt es, über die `compareTo`-Methode für Klassen eine Sortierreihenfolge zu definieren.

Die hier vorgestellte Implementierung ist nicht optimal:

Zum einen sollte man seit Java 1.5 generische Typen verwenden, d. h. statt `Comparable` lieber die generische Variante `Comparable<Telefonnummer>` implementieren. Generics werden aber erst später eingeführt, sodass sie hier noch nicht verwendet werden konnten.

Weiterhin ist das Speichern einer Telefonnummer als `int` sicherlich nicht korrekt, wenn es in dem jeweiligen Land führende Nullen in der Telefonnummer gibt, oder wenn die Nummern länger als die mit einem `int` darstellbare Anzahl von Stellen werden kann.

```
Telefonnummer[] nummern = {
       new Telefonnummer(6221, 72643),
       new Telefonnummer(7262, 5558912),
       new Telefonnummer(6221, 71263),
       new Telefonnummer(7261, 5558912) };

Arrays.sort(nummern);

System.out.println(Arrays.asList(nummern).toString());
```

```
[(6221) 71263, (6221) 72643, (7261) 5558912, (7262) 5558912]
```

3.15 Callbacks [70]

```
public interface Funktion {
    public abstract int apply(int o1, int o2);
}

public class Addition implements Funktion {

    public int apply(int o1, int o2) {
        return o1 + o2;
    }
}

public class Subtraktion implements Funktion {

    public int apply(int o1, int o2) {
        return o1 - o2;
    }
}
```



Das Beispiel hier verwendet das Strategy-Pattern.

```
public class Berechnung {

    public static int berechne(int input1, int input2, Funktion funktion) {

        int ergebnis = funktion.apply(input1, input2);
        return ergebnis;
    }

    public static void main(String[] args) {
        Funktion sub = new Subtraktion();
        Funktion add = new Addition();

        System.out.println(berechne(5, 3, sub));
        System.out.println(berechne(1, 7, add));
    }
}
```



Bei einem **Callback** geht es letztendlich darum, einer Methode eine andere Methode zu übergeben, die diese dann aufrufen kann. Da der Aufrufer die Methode mitgibt, ruft die aufgerufene Methode sozusagen zum Aufrufen zurück, daher der Name Callback.

Callback

Methoden sind in Java, anders als in funktionalen Programmiersprachen, keine eigenständigen Konstrukte. Dies bedeutet, man kann eine Methode nicht einzeln übergeben oder referenzieren.

Bis zur Einführung von Lambdas war daher die einzige Möglichkeit eine Methode zu übergeben, der Umweg über ein Objekt, in dessen Klasse sich die gewünschte Methode befand. Um die

nötige Flexibilität zu bekommen, die Methode auszugestalten, ist man üblicherweise wie folgt vorgegangen:

1. Man definiert ein Interface, das eine Callback-Methode mit den gewünschten Parametern und Rückgabetyp enthält. Im Beispiel das Interface `Funktion` mit der Callback-Methode `apply`.
2. Man definiert Empfängermethode einen Parameter vom Typ des Interfaces. Im Beispiel `berechne`.
3. In der Empfängermethode ruft man die Callback-Methode aus dem Interface auf.
4. Der Verwender implementiert das Interface in einer eigenen Klasse und liefert somit seine Implementierung der Callback-Methode. Im Beispiel `Addition` und `Subtraktion`.
5. Der Verwender erzeugt ein Objekt von der Klasse und übergibt es der Empfängermethode. Im Beispiel in der `main`-Methode zu sehen.

3.16 Default-Methoden: Motivation [72]

- Interfaces können nach der Veröffentlichung nicht mehr geändert werden
- Erweiterung führt zu Compiler-Fehlern bei den Verwendern

```
public interface Flieger {
    public abstract void starten();
    public abstract void landen();
}
```

```
public class Flugzeug implements Flieger {
    public void starten() { /* ... */ }
    public void landen() { /* ... */ }
}
```

Ein gängiges Problem bei Interfaces ist, dass man es nach der Veröffentlichung nicht mehr ändern kann. Der Grund liegt darin, dass jede Änderung des Interfaces, vor allem das Hinzufügen von Methoden, dazu führt, dass alle Verwender nicht mehr kompilieren. Deswegen muss man sehr viel Sorgfalt in den Entwurf der Schnittstellen stecken und trotzdem kommt es noch häufig vor, dass man ein Interface trotzdem später anpassen muss. Als mögliche Lösung kam bisher nur infrage, ein neues Interface anzulegen, dass vom ursprünglichen erbt und es erweitert.

Erweitertes Interface

```
public interface Flieger {
    public abstract void starten();
    public abstract void landen();
    public abstract void sinken();
    public abstract void steigen();
```

```
}
```

Fehlermeldung beim Compilieren

```
The type Flugzeug must implement the inherited abstract method
Flieger.sinken()
    Flugzeug.java line 3
```

Das Interface `Flieger` im Beispiel kann man nicht mehr erweitern, ohne alle Verwender (hier z. B. `Flugzeug`) anzupassen. Würde man die Methoden `steigen()` und `sinken()` hinzufügen, käme es zu Fehlern in `Flugzeug`.

3.17 Lösung: Default-Methoden [74]

Interfaces können **Default-Methoden** (default methods) enthalten

Default-Methoden

- Methoden *mit* Implementierung, durch Schlüsselwort `default` gekennzeichnet — **`default`**
- Werden an die Klasse weitergegeben, die das Interface implementiert
- Können von der Klasse mit eigener Implementierung überschrieben werden
- Interfaces mit Default-Methoden heißen **erweiterte Schnittstellen** — **erweiterte Schnittstellen**
- Default-Methoden können `public` oder `private` sein.

```
public interface Flieger {
    public abstract void starten();
    public abstract void landen();
    public default void sinken() {
        System.out.println("sinke");
    }
    public default void steigen() {
        System.out.println("steige");
    }
}
```

Default-Methoden werden manchmal als **virtual extension methods** oder **defender methods** bezeichnet. Die Bezeichnung *defender methods* macht klar, dass es hier darum geht, vorhandene Implementierungen der Interfaces gegen Änderungen der Interfaces zu verteidigen.

virtual extension methods

defender methods

Im vorliegenden Beispiel sind die Methoden `sinken` und `steigen` Default-Methoden. Wenn man das Interface implementiert und keine eigenen Implementierungen angibt, dann wird die Implementierung aus dem Interface übernommen.

Der Hauptgrund, warum man *Default-Methoden* eingeführt hat, waren die Erweiterungen am Collection-API, die für die Lambda-Funktionen nötig waren. Hätte man keine Default-

Methoden eingeführt, wären alle existierenden Verwender des Collection-APIs auf Probleme gestoßen. Collections werden in einem späteren Kapitel noch ausführlich behandelt.

Dass Default-Methoden in einem Interface `private` sein können, wirkt auf den ersten Blick sinnlos, da sie ja so nicht an die implementierende Klasse weitergegeben werden. Man kann sie aber dazu benutzen, die Implementierung von Default-Methoden in einem Interface auf mehrere, kleinere Methoden aufzuteilen und so ein besseres Design zu realisieren.

Beispiel: private Default-Methoden



```
public interface Flieger {
    public abstract void starten();
    public abstract void landen();
    private void aktion(String text) {
        System.out.println(text);
    }
    public default void sinken() {
        aktion("sinke");
    }
    public default void steigen() {
        aktion("steige");
    }
}
```

Bei einer privaten Default-Methode in einem Interface entfällt das Schlüsselwort `default`. Sie sind ab Java 9 verfügbar.



```
Flugzeug flugzeug = new Flugzeug();
flugzeug.starten();
flugzeug.steigen();
flugzeug.sinken();
flugzeug.landen();
```



```
starte
steige
sinke
lande
```

Die Klasse `Flugzeug` verwendet das Interface `Flieger` in seiner ursprünglichen Form und implementiert nur die beiden Methoden `starten()` und `landen()`. Würde man das Interface einfach um zwei weitere abstrakte Methoden `sinken()` und `steigen()` erweitern, würde es einen Compile-Fehler geben, da `Flugzeug` die Methoden nicht implementiert.

Die Verwendung von Default-Methoden erlaubt es nun, das Interface zu erweitern, *ohne* die Klasse `Flugzeug` zu stören. `Flugzeug` „erbt“ jetzt einfach die beiden Methoden `sinken()` und `steigen()` aus dem Interface, sodass man sie ganz normal aufrufen kann.

3.18 Mehrfachvererbung und Default-Methoden [77]

Implementiert Klasse zwei Interfaces mit gleichnamigen Default-Methoden so *muss* sie die Methode überschreiben

```
public interface IA {
    public default void m() { }
}

public interface IB {
    public default void m() { }
}

public class Impl implements IA, IB {
}
```

```
Duplicate default methods named m with the parameters () and () are
inherited from the types IB and IA
    Impl.javal line 3
```

Interfaces erlauben Mehrfachvererbung und mehrfache Implementierung. Wie wird der berühmte **Deadly-Diamond** vermieden? Beim Deadly-Diamond kommen in einer Klasse durch die Mehrfachvererbung zwei Implementierungen einer Methode zusammen, sodass man beim Aufruf der Methode nicht weiß, welche der beiden Implementierungen genutzt werden soll.

Deadly-Diamond

Java löst dies, indem verlangt wird, dass die Klasse, in der die beiden Default-Methoden zusammenkommen diese überschreiben *muss*.

Im vorliegenden Beispiel liegt die Lösung also darin, dass `Impl` die Methode `m()` überschreiben muss.

```
public class Impl implements IA, IB {
    public void m() { }
}
```

3.19 Vererbung von Default-Methoden [78]

Wenn Interface von einem Interface mit Default-Methoden erbt, kann es

- die Methode ignorieren
  → Methode wird vererbt
- die Methode ohne `default` angeben
  → Methode wird normale abstrakte Methode

- die Methode mit `default` angeben
  → Methode wird überschrieben
  → Zugriff auf ursprüngliche Methode mit `Typ.super.Methode`
- abstrakte Methode durch Default-Methoden ersetzen
  → Implementierende Klasse muss sie nicht mehr implementieren

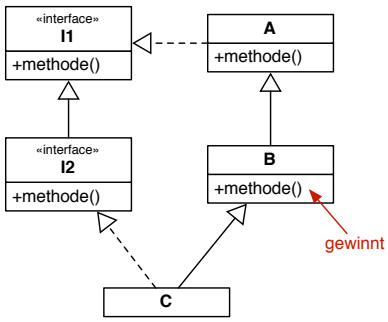
```
public interface ExtFlieger extends Flieger {

    @Override
    public default void steigen() {
        // ersetzt die steigen Methode aus Flieger
        System.out.println("Schnell steigen");
        // Ursprüngliche Methode aufrufen
        Flieger.super.steigen();
    }

    public abstract void sinken(); // Methode ist jetzt abstrakt

    public default void starten() {
        // Methode ist jetzt nicht mehr abstrakt
        System.out.println("Kommt");
    }
}
```

Bei einem Konflikt haben die Methoden einer Klasse Vorrang vor den `default`-Methoden der Interfaces



Deadly-Diamond in Java 8

Das Beispiel lässt sich mit folgendem Source-Code nachstellen.

```
public interface I1 {

    public default void methode() {
        System.out.println("I1");
    }
}

public interface I2 extends I1 {
```

```
    @Override
    public default void methode() {
        System.out.println("I2");
    }
}

public class A implements I1 {

    public void methode() {
        System.out.println("A");
    }
}

public class B extends A {

    @Override
    public void methode() {
        System.out.println("B");
    }
}

public class Main {

    public static void main(String[] args) {
        C c = new C();
        c.methode();
    }
}
```

Die Ausgabe ist dann:

```
B
```



Kapitel 4

Prinzipien für gutes Design

4.1 SOLID-Prinzipien [82]

SOLID-Prinzipien

Die **SOLID-Prinzipien** helfen ein robustes und erweiterbares Software-Design zu entwickeln

- Single Responsibility-Prinzip
- Open/Close-Prinzip
- Liskov Substitution-Prinzip
- Interface Segregation-Prinzip
- Dependency Inversion-Prinzip

Single Responsibility-Prinzip

Single Responsibility-Prinzip: Eine Klasse sollte genau eine Aufgabe haben und alle ihre Dienste sollten genau zu dieser Aufgabe passen

Open/Closed-Prinzip

Open/Closed-Prinzip: Eine Klasse sollte offen für Erweiterungen aber geschlossen für Modifikationen sein

An dieser Stelle muss man verstehen, was genau mit Erweiterung und Modifikation gemeint ist.

Erweiterung

- Bei einer **Erweiterung** leitet man von vorhandenen Klassen neue Klassen ab, die durch geschicktes Überschreiben von Methoden die Funktionalität erweitern. Hierzu gibt es passende Entwurfsmuster, von denen wir bereits einige kennengelernt haben (z. B. Template Pattern).

Modifikation

- Bei einer **Modifikation** wird eine vorhandene Klasse modifiziert, d. h. ihr Quellcode wird verändert.

Einen Verstoß gegen das Open/Closed-Prinzip erkennt man immer daran, wenn man nach dem Anlegen einer neuen Klasse, eine bereits vorher vorhandene Klasse verändert muss.

Liskov Subsititution-Prinzip

Liskov Subsititution-Prinzip: Subklassen sollten echte Subklassen im Sinne des liskovschen Substitutionsprinzips sein

Das liskovsche Substitutionsprinzip wurde bereits weiter oben erläutert.

Interface Segregation-Prinzip

Interface Segregation-Prinzip: Verwender einer Schnittstelle sollten nicht gezwungen werden Interfaces zu implementieren, die sie gar nicht nutzen wollen

Dieses Prinzip besagt, dass Benutzer:innen eines Interfaces nicht mit Methoden konfrontiert werden sollten, die sie nicht benötigen. Jedes Interface sollte sich auf eine einzige Verantwortung beschränken und spezialisiertere Interfaces von einer größeren, allgemeineren Oberklasse abgeleitet werden.

Es ist besser, mehrere kleine, spezialisierte Interfaces zu haben, als ein großes, allgemeines Interface, das viele Methoden enthält, die für bestimmte Klassen nicht relevant sind.

Dependency Inversion-Prinzip: Abstraktionen sollten nicht von Details abhängen, sondern Details von Abstraktionen

Dependency Inversion-Prinzip

Das Dependency Inversion-Prinzip besagt, dass höhere Ebene nicht von darunter liegenden Ebene abhängig sein sollten.

Dies bedeutet, dass man nicht auf konkrete Klassen direkt verweisen sollte, sondern stattdessen auf Abstraktionen, die von diesen Klassen implementiert werden. So kann man eine bessere Trennung zwischen verschiedenen Schichten in einem System erreichen und es wird einfacher, Änderungen in den Implementierungen von Klassen ohne Auswirkungen auf andere Teile des Systems durchzuführen.

Einen Verstoß gegen dieses Prinzip erkennt man daran, dass eine Subtyp in der Schnittstelle eines Supertypen auftaucht. Es wäre also nach diesem Prinzip falsch, wenn in dem Interface `Flieger` eine Klasse auftauchen würde, die das Interface implementiert, z. B. `Superman`.

4.2 STUPID [88]

STUPID-Prinzip

STUPID-Prinzip

Wenn es Prinzipien für gutes Design gibt, so ist es naheliegend, dass es auch einen Gegenmodell gibt, das beschreibt, woran man ein schlechtes Design erkennen kann.

- *Singleton*: Verwendung von Singletons mit globalem Zustand
- *Tight Coupling*: Enge Verzahnung der Komponenten
- *Untestability*: Testen nicht berücksichtigt
- *Premature Optimization*: Unnötige Optimierungen
- *Indescriptive Naming*: Unverständlichen Namen
- *Duplication*: Kopierter Code

Das STUPID-Prinzip ist ein Akronym für „Singleton, Tight Coupling, Untestability, Premature Optimization, Indescriptive Naming“ und Duplication. Es ist ein Begriff aus der Softwareentwicklung, der häufig im Zusammenhang mit der objektorientierten Programmierung verwendet wird, um auf *schlechte* Programmierpraktiken hinzuweisen.

- **Singleton**: Die Anwendung verwendet mehrere Singeltons, die globalen Zustand halten. Durch wird sie schwer testbar und fehleranfällig. Singleton
- **Tight Coupling**: Die Komponenten der Software sind eng miteinander verzahnt und es gibt sehr viele Abhängigkeiten. Tight Coupling

- **Untestability**: Bei der Entwicklung wurde nicht daran gedacht, die Software auch testbar zu gestalten. Es ist schwierig, Unit-Tests zu schreiben und auszuführen. **Untestability**
- **Premature Optimization**: Die Entwickler:innen versuchen, Code zu optimieren, obwohl noch gar nicht feststeht, dass die Optimierung auch notwendig ist. Dies kann zu unnötigem Code und einer unnötigen Komplexität führen. **Premature Optimization**
- **Indescriptive Naming**: Die Naben von Klassen und Methoden ist uneinheitlich und beschreibt die Funktion der jeweiligen Komponenten nicht, oder nur unvollständig. Beispiele sind `var1`, `var2`, `_button1` etc. **Indescriptive Naming**
- **Duplication**: Der Code enthält viele kopierte und damit duplizierte Code-Fragmente. Bei der Fehlerbehebung werden dann regelmäßig Kopien übersehen, sodass die Fehler dort nicht korrigiert werden. **Duplication**

Index

abstrakte Klasse, 8
abstrakte Methode, 8
abstrakten Methoden, 8

Callback, 30

Deadly-Diamond, 34
Default-Methoden, 32
defender methods, 32
Dependency Inversion-Prinzip, 38
Duplication, 39

erweiterte Schnittstellen, 32
Erweiterung, 37

Indescriptive Naming, 39
Interface, 18
Interface Segregation-Prinzip, 37

konkrete Methode, 11

Liskov Subsititution-Prinzip, 37

Mehrfachvererbung, 19
Modifikation, 37

Open/Closed-Prinzip, 37

Premature Optimization, 39

Single Responsibility-Prinzip, 37
Singleton, 38
SOLID-Prinzipien, 37
STUPID-Prinzip, 38

Template Method, 15
Template Method Pattern, 15
Template Method-Pattern, 16
Tight Coupling, 38

Untestability, 39

Vertrag, 12
virtual extension methods, 32

öffentliche Schnittstelle, 18

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Algorithmen



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Algorithmen **1**

1.1 Algorithmus [5] . . . 1
1.2 Algorithmus / Beispiel [8] . . . 1
1.3 Terminierend / Deterministisch [9] . . . 2
1.4 Bausteine für Algorithmen [10] . . . 3

2 Komplexitätstheorie **4**

2.1 Komplexität [12] . . . 4
2.2 Komplexitätstheorie [14] . . . 5
2.3 Fälle [15] . . . 5
2.4 Wachstum von Funktionen [17] . . . 6
2.5 Konstanten [18] . . . 6
2.6 Effekt der Rechner-Performance [19] . . . 7
2.7 O-Notation [20] . . . 7
2.8 Sprechweise [23] . . . 8
2.9 Beispiele für die Komplexität [24] . . . 8
2.10 Θ-Notation [25] . . . 9

3 Suchen und mischen **11**

3.1 Sequentielle Suche [27] . . . 11
3.2 Binäre Suche [29] . . . 12
3.3 Mischverfahren [32] . . . 13
3.4 Direktes Verfahren [33] . . . 13
3.5 Fisher-Yates-Verfahren [35] . . . 14

4 Textsuche **16**

4.1 Brute-Force-Algorithmus [38] . . . 16
4.2 Boyer-Moore-Algorithmus [40] . . . 17
4.3 Boyer-Moore: Verschiebung [41] . . . 18
4.4 Boyer-Moore-Algorithmus am Beispiel [43] . . . 19
4.5 Knuth-Morris-Pratt-Algorithmus [44] . . . 19
4.6 Knuth-Morris-Pratt-Algorithmus am Beispiel [45] . . . 20
4.7 Berechnung der next-Tabelle [46] . . . 21

5 Reguläre Ausdrücke 23
5.1 Zweck [48] . . . 23
5.2 Grundlagen regulärer Ausdrücke [49] . . . 23
5.3 Ersetzungen in Strings [50] . . . 24
5.4 Matcher-Objekt [51] . . . 25
5.5 Anker [52] . . . 26
5.6 Zeichenklassen [54] . . . 26
5.7 Wiederholungen [56] . . . 27
5.8 Greedy Match [58] . . . 28
5.9 Gruppierungen [59] . . . 28
5.10 Gruppierungen in Ersetzungen [60] . . . 29
5.11 Weitere Möglichkeiten [61] . . . 29

6 Sortieren 31
6.1 Einleitung [63] . . . 31
6.2 Insertionsort [64] . . . 31
6.3 Selectionsort [68] . . . 33
6.4 Bubblesort [72] . . . 34
6.5 Mergesort [76] . . . 35
6.6 Quicksort [80] . . . 36
6.7 Verfahren im Vergleich [86] . . . 39

Index ii

Kapitel 1

Algorithmen

1.1 Algorithmus [5]

> Ein *Algorithmus* (algorithm) ist eine allgemeingültige endliche Folge von elementaren Operationen, die maschinell durchgeführt werden können. Die Folge enthält eine Beschreibung, wie diese elementaren Operationen nacheinander durchgeführt werden sollen. Insbesondere ist dabei eine Start- und Ende-Bedingung angegeben.
>
> G. Büchel, Praktische Informatik

> Ein *Algorithmus* ist eine präzise (d. h. in einer festgelegten Sprache abgefasste) endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte.
>
> Saake, Sattler; Algorithmen und Datenstrukturen

> *Algorithmus* (algorithm) – Arbeitsanleitung zum Lösen eines Problems bzw. einer Aufgabe, die so präzise formuliert ist, dass sie von einem Computer ausgeführt werden kann.
>
> Dietrich Boles

- Formulierung von **Algorithmen** — Algorithmen
  - Umgangssprachlich
  - Programmiersprache
  - Aktivitätsdiagramme / Struktogramme etc.
- Ausführung von Algorithmen durch einen Prozessor (Mensch / Computer) → **Prozess** (process) — Prozess

1.2 Algorithmus / Beispiel [8]

Aufgabe: Berechne die Summe der Zahlen von 1 bis n

Umgangssprachliche Formulierung

```
Addiere für eine vorgegebene natürliche Zahl n
die Zahlen von 1 bis n. Dies ist das Resultat.
```

Programmiersprache

```
int n = readInt();
int erg = 0;
int aktZahl = 1;
while (aktZahl <= n) {
    erg = erg + aktZahl ;
    aktZahl = aktZahl + 1;
}
printInt(erg);
```

1.3 Terminierend / Deterministisch [9]

Ein Algorithmus heißt

- **terminierend**, wenn er zu einem Ende kommt — terminierend
- **determiniert**, wenn er immer zum selben Ergebnis kommt — determiniert
- **deterministisch**, wenn er immer dieselben Schritte durchläuft — deterministisch

Jeder deterministische Algorithmus ist auch determiniert

Wichtigste Klasse: deterministische, terminierende Algorithmen

Terminierender Algorithmus

Ein *terminierender Algorithmus* ist ein Algorithmus, der immer ein Ende erreicht und ein Ergebnis liefert. Dies bedeutet, dass der Algorithmus unter allen möglichen Eingabebedingungen und Zuständen immer aufhört zu arbeiten und ein Ergebnis liefert, anstatt in eine Endlosschleife zu geraten oder in einen unerwarteten Zustand zu gelangen.

Ein anderes Beispiel wäre ein Algorithmus zum Suchen eines bestimmten Werts in einem Array. Der Algorithmus wird das Array elementweise durchlaufen und jeden Wert mit dem gesuchten Wert vergleichen. Sobald der gesuchte Wert gefunden wird, gibt der Algorithmus den Index des Werts im Array zurück und beendet seine Arbeit.

Ein Algorithmus, der nicht terminierend ist, wäre ein Algorithmus, der in einer Endlosschleife stecken bleibt und niemals aufhört zu arbeiten. Ein Beispiel wäre ein Algorithmus, der versucht, eine unendliche Summe von 1 zu berechnen, und niemals aufhört zu arbeiten.

Determinierter Algorithmus

Ein *determinierter Algorithmus* ist ein Algorithmus, dessen Ausführung immer zu dem gleichen Ergebnis führt, unabhängig von der spezifischen Implementierung oder der Hardware, auf der

er ausgeführt wird. Dies bedeutet, dass der Algorithmus immer das gleiche Ergebnis liefert, wenn er mit den gleichen Eingabedaten aufgerufen wird.

Ein Beispiel für einen determinierten Algorithmus ist das Sortieren eines Arrays. Der Algorithmus wird das Array elementweise durchlaufen und die Elemente an ihre korrekten Positionen sortieren. Einmal sortiert, gibt der Algorithmus das sortierte Array zurück und beendet seine Arbeit. Wenn der Algorithmus mit den gleichen Eingabedaten aufgerufen wird, wird er immer das gleiche sortierte Array zurückgeben.

Es ist wichtig, dass ein Algorithmus determiniert ist, da dies bedeutet, dass das Ergebnis reproduzierbar und verlässlich ist.

Beispiel: determinierter, nicht deterministischer Algorithmus

1. Nehmen Sie eine Zahl x ungleich 0.
2. Entweder: Rechnen Sie (3x + x)/x
   Oder: Rechnen Sie x - (x - 4)
3. Schreiben Sie das Ergebnis auf

Beispiel: nichtdeterminierter Algorithmus:

1. Nehmen Sie eine beliebige Zahl x.
2. Addieren Sie 5 hinzu und multiplizieren Sie mit 3.
3. Schreiben Sie das Ergebnis auf.

Deterministischer Algorithmus

Ein *deterministischer Algorithmus* ist ein Algorithmus, bei dem nur definierte und reproduzierbare Zustände auftreten. Für die gleiche Eingabe folgt auch immer die gleiche Ausgabe und zusätzlich wird die gleiche Folge an Zuständen durchlaufen. Zu jedem Zeitpunkt ist der nachfolgende Abarbeitungsschritt des Algorithmus eindeutig festgelegt.[1] Das bedeutet auch, dass alle Zwischenergebnisse innerhalb des Algorithmus immer gleich sind.

1.4 Bausteine für Algorithmen [10]

- Elementare Operationen
- Sequenzielle Ausführung
- Parallele Ausführung
- Bedingte Ausführung
- Schleife
- Unterprogramm
- Rekursion

Kapitel 2

Komplexitätstheorie

2.1 Komplexität [12]

In der Informatik spielt die Analyse der Laufzeit und Speicherbedarf von Algorithmen eine wichtige Rolle. Die **Komplexitätstheorie** ist ein Teilgebiet der Informatik, das sich mit der Untersuchung der Leistungsfähigkeit von Algorithmen beschäftigt.

Komplexitätstheorie

Sie verwendet verschiedene Methoden, um die Laufzeit von Algorithmen zu beschreiben und zu vergleichen. Diese Methoden umfassen die Bestimmung der Anzahl der notwendigen Schritte (oder der notwendigen Rechenzeit), die Untersuchung der Anzahl der notwendigen Speicherzellen und die Analyse des Verhaltens des Algorithmus in Bezug auf die Größe der Eingabe.

In diesem Kapitel werden wir uns mit verschiedenen Aspekten der Komplexitätstheorie beschäftigen.

Effizienzbestimmung von Algorithmen

- Rechenzeit
- belegter Speicher
- Eingabe/Ausgabe (mit Wirkung auf Laufzeit)
- Hauptspeicherzugriffe
- Externer Speicher (um Faktor 100 langsamer)

Faktoren

- Leistungsfähigkeit der Hardware
- Qualität des vom Compiler generierten Codes
- Eingabe (Anzahl der Eingabewerte n)
- Zeitkomplexität des Algorithmus ⇒ Laufzeit als Funktion der Eingabegröße ausgedrückt: T(n)

Die Komplexität von Algorithmen beschreibt, wie schnell die Laufzeit oder der Speicherbedarf eines Algorithmus mit der Größe der Eingabe wächst. Es gibt verschiedene Möglichkeiten, die Komplexität von Algorithmen zu messen, wie zum Beispiel die Anzahl der ausgeführten

Operationen oder die Anzahl der Zugriffe auf den Speicher. Ein wichtiger Bestandteil der Algorithmenanalyse ist die Bestimmung der Laufzeitkomplexität, die normalerweise in der asymptotischen Notation angegeben wird. Ein Beispiel für einen Algorithmus mit niedriger Komplexität ist eine lineare Suche, die in $\mathcal{O}(n)$ Zeit abläuft, während ein Beispiel für einen Algorithmus mit hoher Komplexität eine sortierte Suche ist, die in $\mathcal{O}(n \log n)$ Zeit abläuft. Es ist wichtig, die Komplexität von Algorithmen zu verstehen, um die Leistung von Programmen zu optimieren und die richtigen Algorithmen für bestimmte Aufgaben auszuwählen.

2.2 Komplexitätstheorie [14]

- Komplexitätstheorie befasst sich mit
  - **Zeitkomplexität** (Aufwand an Rechenzeit) — Zeitkomplexität
  - **Speicherkomplexität** (Aufwand an Speicherplatz) — Speicherkomplexität
- Abstraktes Maschinenmodell zugrundeliegend
  - Turingmaschine, Random-Access-Machine
  - Elementaroperationen (z. B. Zuweisung, Vergleich, arithmetische Operationen, Array-Zugriff)
  - Nicht-elementare Operationen (z. B. Schleife, Prozeduraufruf)
- Komplexitätsklasse und asymptotisches Verhalten
  - von abstrakter Eingabegrößen abhängig

2.3 Fälle [15]

- T_{best}
  Bester Fall (**best case**) — best case
- T_{worst}
  Schlimmster Fall (**worst case**) — worst case
- T_{avg}
  Durchschnittlicher Fall (**average case**) — average case

In der Komplexitätstheorie beschreiben Bester Fall (**best case**), Durchschnittlicher Fall (**average case**) und Schlimmster Fall (**worst case**) die verschiedenen Szenarien, in denen ein Algorithmus aufgerufen werden kann. — best case, average case, worst case

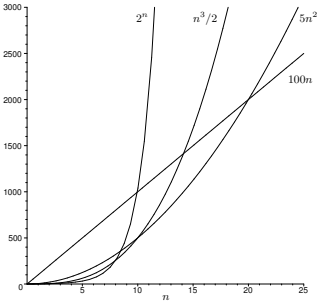
1. *Best case* bezieht sich auf das Szenario, in dem der Algorithmus am schnellsten ausgeführt wird. Dies kann zum Beispiel der Fall sein, wenn die Eingabe bereits sortiert ist und der Algorithmus diese Eigenschaft nutzen kann.
2. *Average case* bezieht sich auf das durchschnittliche Szenario, in dem der Algorithmus aufgerufen wird. Dies kann sich auf die durchschnittliche Laufzeit des Algorithmus oder auf die durchschnittliche Anzahl von Schritten beziehen, die der Algorithmus benötigt, um zu einem Ergebnis zu gelangen.

3. *Worst case* bezieht sich auf das Szenario, in dem der Algorithmus am langsamsten ausgeführt wird. Dies kann zum Beispiel der Fall sein, wenn die Eingabe in der schlechtesten möglichen Reihenfolge vorliegt und der Algorithmus diese Eigenschaft nicht nutzen kann.

```java
boolean search(int s, int[] data) {
    for (int i = 0; i < data.length; i++) {
        if (data[i] == s) {
            return true;
        }
    }
    return false;
}
```

- T_{best}: `c` ist das erste Element
- T_{worst}: `c` ist nicht im Array enthalten
- T_{avg}: `c` ist mit einer gewissen Wahrscheinlichkeit im Array enthalten

2.4 Wachstum von Funktionen [17]



Quelle: Markus Nebel Sebastian Wild: Entwurf und Analyse von Algorithmen, 2. Auflage

2.5 Konstanten [18]

- Verhalten bei großen `n` ist von Interesse
- Terme mit niedrigem Grad fallen wenig ins Gewicht

⇒ Wir betrachten nur die Terme mit dem höchsten Grad

n	1	2	3	10	100	1000	10000
T_1	3	7	13	111	10101	1001001	100010001
T_2	1	4	9	100	10000	1000000	100000000

$$T_1(n) = n^2 + n + 1$$
$$T_2(n) = n^2$$
$$T_1 \approx T_2$$

2.6 Effekt der Rechner-Performance [19]

- Effekt der Performance nimmt mit wachsender Zeitkomplexität ab
- Bei z. B. 2^n ist es unerheblich, ob der Rechner 1000 Mal schneller ist $2^{100} \approx 2^{90}$
- Wichtig ist die *Größenordnung* der Zeitkomplexität
- Wir suchen die **obere Schranke** der Zeitkomplexität für große n

obere Schranke

In der asymptotischen Notation wird die Laufzeitkomplexität eines Algorithmus in Bezug auf die Größe der Eingabe angegeben.

2.7 O-Notation [20]

O-Notation oder **Landau-Notation** beschreibt die *obere Schranke* der Laufzeitkomplexität für große n

O-Notation
Landau-Notation

- Seien $f : \mathbb{N} \to \mathbb{R}$ und $g : \mathbb{N} \to \mathbb{R}$ Funktionen
- $\mathcal{O}(g) = \{f | \exists\, n_0 \in \mathbb{N}, c \in \mathbb{R}, c > 0 : \ \forall\, n \geq n_0 : |f(n)| \leq c \cdot |g(n)|\}$
- $\mathcal{O}(g)$ ist die Menge aller Funktionen f, die höchstens so stark wachsen, wie g
- Falls $f \in \mathcal{O}(g)$, dann ist f durch g *nach oben beschränkt*

Rechenregeln in der O-Notation

- $\mathcal{O}(\log_a n) = \mathcal{O}(\log n)$
- $\mathcal{O}(k \cdot g(n)) = \mathcal{O}(g(n))$
- $\mathcal{O}(n^k + n^{k-1} + ...) = \mathcal{O}(n^k)$

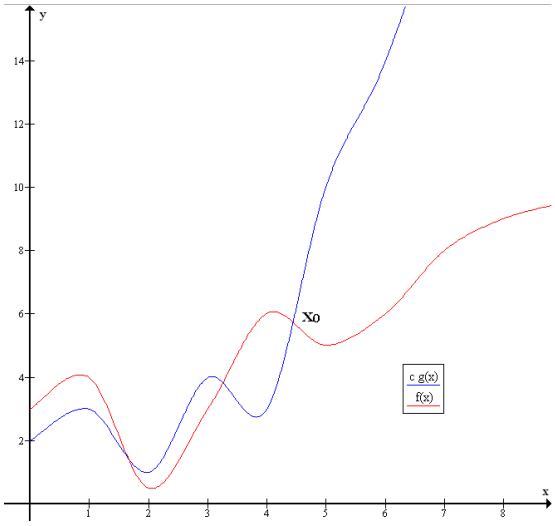
Die Landau-Notation, auch als Big-O-Notation bekannt, ist eine Methode zur Schätzung der asymptotischen Laufzeitkomplexität von Algorithmen. Sie gibt an, wie schnell die Laufzeit oder der Speicherbedarf eines Algorithmus mit der Größe der Eingabe wächst, indem sie den dominanten Faktor des Wachstums beschreibt.

In der Landau-Notation wird die Laufzeitkomplexität eines Algorithmus durch eine Funktion von der Größe der Eingabe n beschrieben. Beispielsweise ist die Laufzeitkomplexität eines Algorithmus, der in einer Schleife n-mal ausgeführt wird, in $\mathcal{O}(n)$.

Das $\mathcal{O}$ steht hierbei für die Landau-Notation und gibt an, wie schnell die Laufzeit des Algorithmus mit der Größe der Eingabe (in der Regel durch die Variable n repräsentiert) wächst.

Es ist wichtig zu beachten, dass die Landau-Notation nur die Wachstumsrate beschreibt und nicht die tatsächliche Laufzeit. Ein Algorithmus mit $\mathcal{O}(n)$ Laufzeitkomplexität kann bei kleinen Eingabegrößen schneller sein als ein Algorithmus mit $\mathcal{O}(\log n)$ Laufzeitkomplexität, wenn der konstante Faktor des Algorithmus mit niedriger Komplexität höher ist.

Da die O-Notation die obere Schranke betrachtet, handelt es sich um eine *Worst-Case-Abschätzung*.



Beispiel für O-Notation

2.8 Sprechweise [23]

Für einige Komplexitätsklassen haben sich feststehende Begriffe eingebürgert

$\mathcal{O}$	**sprechweise**
$\mathcal{O}(1)$	konstant
$\mathcal{O}(\log n)$	logarithmisch
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	n log n
$\mathcal{O}(n^2)$	quadratisch
$\mathcal{O}(n^3)$	kubisch
$\mathcal{O}(n^k)$	polynomiell
$\mathcal{O}(2^n)$	exponentiell

2.9 Beispiele für die Komplexität [24]

Komplexität	**Beispiel**
$f \in \mathcal{O}(1)$	Feststellen, ob eine Binärzahl gerade ist
$f \in \mathcal{O}(\log n)$	Binäre Suche im sortierten Feld mit n Einträgen
$f \in \mathcal{O}(\sqrt{n})$	Anzahl der Divisionen des naiven Primzahltests
$f \in \mathcal{O}(n)$	Suche im unsortierten Feld mit n Einträgen (Bsp. Lineare Suche)
$f \in \mathcal{O}(n \log n)$	Vergleichbasierte Algorithmen zum Sortieren von n Zahlen, Mergesort, Heapsort
$f \in \mathcal{O}(n^2)$	Einfache Algorithmen zum Sortieren von n Zahlen, Selectionsort
$f \in \mathcal{O}(n^m)$	Einfache Algorithmen
$f \in \mathcal{O}(2^n)$	Erfüllbarkeitsproblem der Aussagenlogik (SAT) mittels erschöpfender Suche
$f \in \mathcal{O}(n!)$	Problem des Handlungsreisenden (mit erschöpfender Suche)

1. *Lineare Suche*: Ein Algorithmus, der durch eine Liste von Elementen iteriert, um ein bestimmtes Element zu finden. Dieser Algorithmus hat eine Laufzeitkomplexität von

$\mathcal{O}(n)$, da er durch die Liste geht und die Anzahl der Iterationen direkt proportional zur Größe der Eingabe (n) ist.

2. *Binäre Suche*: Ein Algorithmus, der in einer sortierten Liste von Elementen nach einem bestimmten Element sucht. Dieser Algorithmus hat eine Laufzeitkomplexität von $\mathcal{O}(\log n)$, da er die Liste jedes Mal in der Hälfte teilt, wenn er das gesuchte Element nicht gefunden hat und da die Anzahl der Iterationen logarithmisch zur Größe der Eingabe (n) ist.
3. *Bubble sort*: Ein Algorithmus zum Sortieren von Elementen in einer Liste, indem er benachbarte Elemente vergleicht und vertauscht. Dieser Algorithmus hat eine Laufzeitkomplexität von $\mathcal{O}(n^2)$, da er durch die Liste geht und für jeden Durchlauf eine Anzahl von Vergleichen und Austauschen proportional zur Größe der Eingabe (n) durchführt.
4. *Quicksort*: Ein Algorithmus zum Sortieren von Elementen in einer Liste, indem er ein Element als Pivot wählt und die Liste in Elemente kleiner und größer als das Pivot aufteilt, und diese Schritte rekursiv durchführt. Dieser Algorithmus hat eine Laufzeitkomplexität von $\mathcal{O}(n \log n)$, da er durch die Liste geht und für jeden Durchlauf eine Anzahl von Vergleichen und Austauschen proportional zur Größe der Eingabe (n) durchführt und die Anzahl der Durchläufe logarithmisch zur Größe der Eingabe (n) ist.

2.10 Θ-Notation [25]

Θ-Notation liefert die *asymptotisch scharfe Schranke* für einen Algorithmus Θ-Notation

- Seien $f : \mathbb{N} \to \mathbb{R}$ und $g : \mathbb{N} \to \mathbb{R}$ Funktionen
- $\Theta(g) = \{f | \exists\, n_0 \in \mathbb{N}, \exists c_1, c_2 \in \mathbb{R} : \ \forall\, n \geq n_0 : c_1 \cdot g(g) \leq f(n) \leq c_2 \cdot g(n)\}$
- $\Theta(g)$ ist die Menge aller Funktionen f, die genauso stark wachsen, wie g

Die Θ-Notation wird in der Praxis selten verwendet

Die Theta-Notation (Θ-Notation) ist eine weitere Methode zur Beschreibung der Laufzeitkomplexität von Algorithmen. Sie gibt an, wie schnell die Laufzeit eines Algorithmus im Durchschnitt mit wachsendem Eingabegröße zunimmt.

Die Θ-Notation beschreibt die Schätzung der Laufzeitkomplexität eines Algorithmus durch eine obere und eine untere Schranke. Ein Algorithmus hat eine Laufzeit von Θ(f(n)), wenn es eine Konstante c_1 und c_2 und eine Eingabegröße n_0 gibt, sodass für alle Eingabegrößen $n > n_0$ gilt: $c_1 \cdot f(n) \leq$ Laufzeit des Algorithmus $\leq c_2 \cdot f(n)$

Das bedeutet, dass die tatsächliche Laufzeit des Algorithmus im Durchschnitt von der angegebenen Funktion f(n) abhängt.

Die Theta-Notation ist also enger als die Big-O-Notation, die nur die obere Schranke der Laufzeit angibt. Sie ermöglicht es uns, die durchschnittliche Laufzeit eines Algorithmus genauer zu beschreiben.

Wie die Big-O-Notation gibt Theta-Notation auch keine Auskunft über die tatsächliche Laufzeit des Algorithmus. Es ist nur eine Schätzung der durchschnittlichen Laufzeitkomplexität.

Die Θ-Notation hat in der Praxis keine große Bedeutung, weil es sehr schwierig ist, die Funktion g(n) nur schwer finden lässt.

Kapitel 3

Suchen und mischen

3.1 Sequentielle Suche [27]

Die sequentielle Suche, auch als lineare Suche bekannt, ist ein Algorithmus zum Suchen eines bestimmten Werts in einem Array. Der Algorithmus durchläuft das Array elementweise, beginnend beim ersten Element, und vergleicht jeden Wert mit dem gesuchten Wert. Wenn der gesuchte Wert gefunden wird, wird die Suche beendet und der Index des Werts im Array zurückgegeben. Wenn der gesuchte Wert nicht im Array enthalten ist, gibt die Suche eine Fehlermeldung oder einen speziellen Wert (-1) zurück.

Sequentielle Suche Sequentielle Suche

- Lineares Durchlaufen der Folge
- Funktioniert auch bei unsortierten Folgen

```
public int search(int e, int[] data) {
    for (int i = 0; i < data.length; i++) {
        if (e == data[i]) {
            return i;
        }
    }

    return NOT_FOUND;
}
```

Die sequentielle Suche ist ein einfacher und verständlicher Algorithmus, der jedoch ineffizient sein kann, da er das Array elementweise durchläuft. Daher hat die sequentielle Suche eine Komplexität von $\mathcal{O}(n)$, wodurch sie langsamer als eine binäre Suche ($\mathcal{O}(\log n)$) ist, es sei denn das Array ist klein.

Für nicht sortierte Arrays ist die sequentielle Suche das einzige funktionierende Suchverfahren.

Aufwand

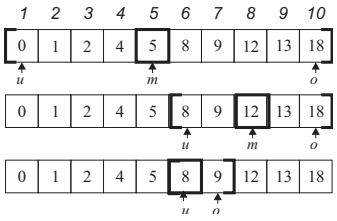
Fall	Vergleiche
bester Fall	1
schlechtester Fall	n
Durchschnitt (erfolgreiche Suche)	(n + 1)/2
Durchschnitt (erfolglose Suche)	n

3.2 Binäre Suche [29]

Die binäre Suche ist ein Algorithmus zum Suchen eines bestimmten Werts in einem sortierten Array. Der Algorithmus funktioniert durch das Teilen des Arrays in immer kleinere Teile, bis der gesuchte Wert gefunden wird oder bis feststeht, dass der Wert nicht im Array enthalten ist.

Binäre Suche Binäre Suche

- Folgt dem Prinzip „teile und herrsche“
- Funktioniert bei einer sortierten Folge
- deutlich schneller als lineare Suche



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Der Algorithmus beginnt damit, den mittleren Wert des Arrays zu untersuchen. Wenn der gesuchte Wert gleich dem mittleren Wert ist, wird die Suche beendet und der Index des Werts im Array zurückgegeben. Wenn der gesuchte Wert kleiner als der mittlere Wert ist, wird die Suche im linken Teil des Arrays fortgesetzt. Ist er größer, wird die Suche im rechten Teil des Arrays fortgesetzt.

Der Prozess wiederholt sich, bis der gesuchte Wert gefunden wurde oder bis der Suchbereich auf eine leere Liste reduziert wurde, was bedeutet, dass der gesuchte Wert nicht im Array enthalten ist.

```java
public int search(int e, int[] data) {
    int u = 0;
    int o = data.length - 1;
    while (u <= o) {
        int m = (u + o) / 2;
        if (data[m] == e) {
            return m;
        }
        else if (data[m] > e) {
            o = m - 1;
        }
```

```
        else {
            u = m + 1;
        }
    }

    return NOT_FOUND;
}
```

Die binäre Suche ist ein sehr effizienter Algorithmus, da er die Anzahl der zu untersuchenden Werte mit jedem Schritt halbiert. Im Allgemeinen hat die binäre Suche eine Komplexität von $\mathcal{O}(\log n)$, wodurch sie schneller als eine sequentielle Suche ($\mathcal{O}(n)$) ist. Es muss jedoch darauf geachtet werden, dass das Array sortiert sein muss, damit die binäre Suche funktioniert.

Aufwand

Fall	Vergleiche
bester Fall	1
schlechtester Fall	$\log_2(n)$
Durchschnitt (erfolgreiche Suche)	$\log_2(n)$
Durchschnitt (erfolglose Suche)	$\log_2(n)$

3.3 Mischverfahren [32]

Ein **Mischverfahren** Mischverfahren

- sollte das ursprüngliche Array *vollständig* durchmischen
- sollte das ursprüngliche Array *zufällig* durchmischen
- sollte *effizient* sein, d. h. möglichst wenig Schritte benötigen
- sollte *einfach* zu implementieren und zu verstehen sein

3.4 Direktes Verfahren [33]

Bei einem direkten Ansatz wird zunächst eine aus den Zahlen von 1 bis n bestehende Liste betrachtet. Nach einer Ziehung einer Zufallszahl zwischen 1 und n wird das entsprechende Listenelement als die erste Zahl der Ergebnispermutation notiert und dieses Element anschließend aus der Liste gestrichen. Im nächsten Schritt wird dann eine uniform verteilte Zufallszahl zwischen 1 und n-1 gezogen, das entsprechende Listenelement als zweite Zahl der Ergebnispermutation notiert und dieses Element ebenfalls wieder gestrichen. Dieses Vorgehen wird fortgesetzt, bis schließlich keine Zahl mehr in der Liste vorhanden ist.

Komplexität: $\mathcal{O}(n^2)$

```
function randperm(n)
  P = zeroes(n)          // Ergebnispermutation mit Nullen ↩
      initialisieren
```

```
  N = [1:n]              // Feld mit den Zahlen von 1 bis n
  for i = 1 to n         // Schleife über die Einträge von P
    z = random(n-i+1)    // Gleichverteilte Zufallszahl zwischen 1 ↩
        und n-i+1
    P(i) = N(z)          // Wähle als nächstes die z-te ↩
        verbleibende Zahl
    N = setdiff(N,N(z))  // Entferne diese Zahl aus den ↩
        verbleibenden Zahlen
  end
  return P
```

```java
public void shuffle(int[] data) {
    Random rand = new Random();
    int n = data.length;
    int[] result = new int[n];
    var index = new LinkedList<Integer>();

    for (int i = 0; i < n; i++) {
        index.add(i);
    }

    for (int i = 0; i < n; i++) {
        int z = rand.nextInt(n - i);
        result[i] = data[index.get(z)];
        index.remove(z);
    }

    System.arraycopy(result, 0, data, 0, n);
}
```

3.5 Fisher-Yates-Verfahren [35]

Fisher-Yates-Verfahren
Knuth-Shuffle

Das **Fisher-Yates-Verfahren** (auch als **Knuth-Shuffle** bekannt) ist ein Algorithmus zum Mischen von Daten in einem Array oder einer Liste. Der Algorithmus funktioniert in-place, d. h. er benötigt keinen zusätzlichen Speicherplatz.

Fisher-Yates-Algorithmus ist ein sicheres und effizientes Verfahren für das Mischen von Daten und wird häufig in zufälligen Sampling-Anwendungen und Spiele verwendet.

Komplexität: $\mathcal{O}(n)$

```
function randperm(n)
  P = [1:n]              // Initialisierung mit der identischen ↩
      Permutation
  for i = n downto 2     // Schleife über die Einträge von P
    z = random(i)        // Gleichverteilte Zufallszahl 1 <= z <= i
    swap(P(i), P(z))     // Vertausche die Zahlen an den Stellen i ↩
        und z
```

```
  end
  return P
end
```

Der Algorithmus beginnt an der letzten Stelle des Arrays und wählt dort einen zufälligen Index zwischen 0 und der aktuellen Position aus. Der Wert an der aktuellen Position wird dann mit dem Wert an dem zufällig ausgewählten Index getauscht. Dieser Vorgang wird für jede Position im Array von hinten nach vorn wiederholt. Auf diese Weise werden die Werte im Array zufällig gemischt.

```
private static void shuffle(int[] data) {
    Random rand = new Random();

    for (int i = 0; i < data.length-1; i++) {
        int randomIndexToSwap = rand.nextInt(data.length);
        int temp = data[randomIndexToSwap];
        data[randomIndexToSwap] = data[i];
        data[i] = temp;
    }
}
```

Kapitel 4

Textsuche

4.1 Brute-Force-Algorithmus [38]

Der *Brute-Force-Algorithmus* ist ein einfacher Ansatz zur Textsuche, bei dem jedes Zeichen des gesuchten Musters mit jedem Zeichen des Texts verglichen wird. Der Algorithmus überprüft jede mögliche Übereinstimmung des Musters mit dem Text, indem er das Muster Zeichen für Zeichen mit dem Text vergleicht, beginnend von der ersten Stelle. Wenn eine Übereinstimmung gefunden wird, wird die nächste Stelle des Musters und des Texts verglichen. Dieser Prozess wird fortgesetzt, bis entweder das komplette Muster gefunden wurde oder es eine Abweichung zwischen dem Muster und dem Text gibt.

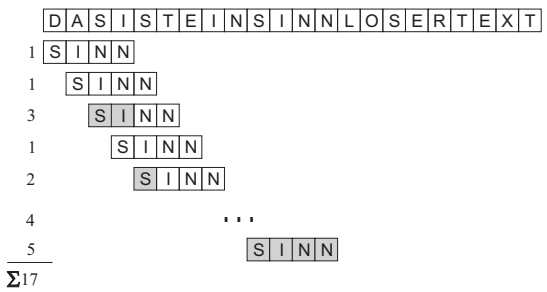
Der Brute-Force-Algorithmus ist einfach zu implementieren und zu verstehen, aber seine Laufzeitkomplexität ist $\mathcal{O}(n \cdot m)$, wobei n die Länge des Texts und m die Länge des Musters ist. Daher kann der Algorithmus bei großen Texten und Mustern sehr langsam sein.

Um die Effizienz zu verbessern, gibt es andere, schnellere Algorithmen, wie den Knuth-Morris-Pratt-Algorithmus oder den Boyer-Moore-Algorithmus.

Brute-Force-Algorithmus

Brute-Force-Algorithmus

- vergleicht jedes Zeichen der Suche mit jedem Zeichen des Textes
- Einfach zu implementieren und zu verstehen
- Komplexität: $\mathcal{O}(n \cdot m)$



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

```
public int search(String haystack, String needle) {
```

```
    int n = haystack.length();
    int m = needle.length();

    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (haystack.charAt(i + j) != needle.charAt(j)) {
                break;
            }
        }
        if (j == m) {
            return i;
        }
    }
    return NOT_FOUND;
}
```

4.2 Boyer-Moore-Algorithmus [40]

Boyer-Moore-Algorithmus

Der **Boyer-Moore-Algorithmus** ist ein effizienter Algorithmus zur Suche von Mustern in Texten. Er basiert auf zwei Techniken: dem Bad-Character-Heuristik und dem Good-Suffix-Heuristik.

Die Bad-Character-Heuristik nutzt die Tatsache aus, dass bei einer Nichtübereinstimmung zwischen dem Suchmuster und dem Text, das nicht übereinstimmende Zeichen im Text dazu verwendet werden kann, um den Vergleich fortzusetzen, ohne dass der Text erneut von vorne gelesen werden muss.

Die Good-Suffix-Heuristik nutzt die Tatsache aus, dass bei einer Nichtübereinstimmung zwischen dem Suchmuster und dem Text, Teile des Suchmusters, die bereits übereinstimmten, dazu verwendet werden können, um den Vergleich fortzusetzen, ohne dass das Suchmuster erneut von vorne gelesen werden muss.

Der Algorithmus arbeitet von hinten nach vorn und vergleicht das Suchmuster mit dem Text, anstatt von vorn nach hinten zu arbeiten, wie es bei anderen Algorithmen der Fall ist. Dies ermöglicht es, die Bad-Character-Heuristik und die Good-Suffix-Heuristik effektiver anzuwenden.

Der Algorithmus berechnet zunächst die Skip-Tabellen für die Bad-Character-Heuristik und die Good-Suffix-Heuristik, die verwendet werden, um die Menge der Zeichen im Text und im Suchmuster zu überspringen, die bei einer Nichtübereinstimmung verglichen werden müssen. Dann wird der Algorithmus die Zeichen im Text von hinten nach vorn vergleichen und die Skip-Tabellen verwenden, um die Menge der Zeichen zu überspringen, die bei einer Nichtübereinstimmung verglichen werden müssen. Wenn eine Übereinstimmung gefunden wird, wird der Algorithmus das nächste Zeichen im Text und im Suchmuster vergleichen, bis entweder das gesamte Suchmuster im Text gefunden wurde oder alle Zeichen im Text verglichen wurden.

Der Boyer-Moore-Algorithmus ist in der Praxis einer der schnellsten Algorithmen für die Textsuche, insbesondere bei großen Texten und langen Suchmustern.

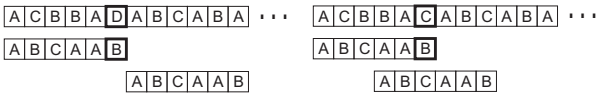
Boyer-Moore-Algorithmus

Boyer-Moore-Algorithmus

- sehr schneller Suchalgorithmus
- führt Vorverarbeitung des Suchstrings durch
- gut geeignet für die Suche von kurzen Strings in langen Texten $\mathcal{O}(n + m)$
- verschiebt das Suchfenster in größeren Schritten

4.3 Boyer-Moore: Verschiebung [41]

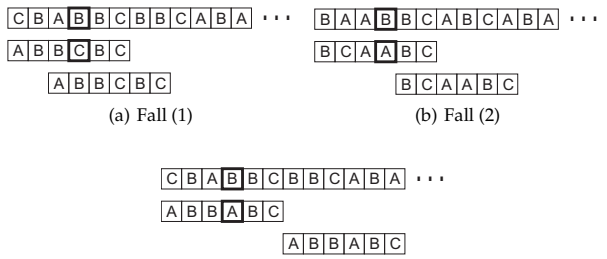
Verschiebung bei Nichtübereinstimmung I



(a) Zeichen kommt im Muster nicht vor (b) Zeichen kommt im Muster vor

Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Verschiebung bei Nichtübereinstimmung II



(a) Fall (1) (b) Fall (2)

Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

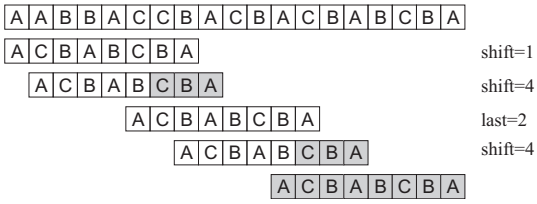
```java
public int search(String haystack, String needle) {
    char[] text = haystack.toCharArray();
    char[] pattern = needle.toCharArray();

    int n = text.length;
    int m = pattern.length;

    if (m == 0) return 0; // Test for empty string
    // Initialization, create Map of last position of each character = O(n)
    Map<Character, Integer> last = new HashMap<>();
    for (char c : text) {
        // set all chars, by default, to -1
        last.put(c, -1);
    }
    for (int i = 0; i < m; i++) {
        // update last seen positions
        last.put(pattern[i], i);
```

```
    }
    //Start with the end of the pattern aligned at index m-1 in the text.
    //index into the text
    int i = m - 1;
    // index into the pattern
    int k = m - 1;
    while (i < n) {
        if (text[i] == pattern[k]) {
            // match! return i if complete match; otherwise, keep checking
            if (k == 0) {
                return i;
            }
            i--; k--;
        }
        else { // jump step + restart at end of pattern
            //iterate over text
            i += m - Math.min(k, 1 + last.get(text[i]));
            //move to end of pattern
            k = m - 1;
        }
    }
    // not found
    return NOT_FOUND;
}
```

4.4 Boyer-Moore-Algorithmus am Beispiel [43]



c	A	B	C	D
last[*i*]	7	6	5	-1

i	0	1	2	3	4	5	6	7	8
pat[*i*]	A	C	B	A	B	C	B	A	
suffix[*i*]	1	0	0	3	0	0	0	8	0
shift[*i*]	7	7	7	7	4	7	7	1	0

Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

4.5 Knuth-Morris-Pratt-Algorithmus [44]

Knuth-Morris-Pratt

Der **Knuth-Morris-Pratt** (KMP) Algorithmus ist ein effizienter Algorithmus zur Suche von Mustern in Texten. Es basiert auf der Erstellung eines „failure-array“ (Fehlerarray), das Informationen über die übereinstimmenden Teilmuster im Suchmuster enthält. Dies ermöglicht

es dem Algorithmus, schnell und effizient die Position im Text fortzusetzen, an der die Übereinstimmung fortgesetzt werden kann, anstatt erneut von vorne zu beginnen, wenn eine Nichtübereinstimmung auftritt.

Der Algorithmus beginnt damit, das failure-array zu berechnen, indem er jedes Teilmuster im Suchmuster vergleicht und die Länge des übereinstimmenden Präfixes und Suffixes ermittelt. Dieses failure-array wird dann verwendet, um schnell die Position im Text fortzusetzen, an der die Übereinstimmung fortgesetzt werden kann, wenn eine Nichtübereinstimmung auftritt.

Der Algorithmus vergleicht dann die Zeichen im Text mit denen im Suchmuster von vorne nach hinten und verwendet das failure-array, um die Position im Text fortzusetzen, an der die Übereinstimmung fortgesetzt werden kann, wenn eine Nichtübereinstimmung auftritt. Wenn eine Übereinstimmung gefunden wird, wird das nächste Zeichen im Text und im Suchmuster verglichen, bis entweder das gesamte Suchmuster im Text gefunden wurde oder alle Zeichen im Text verglichen wurden.

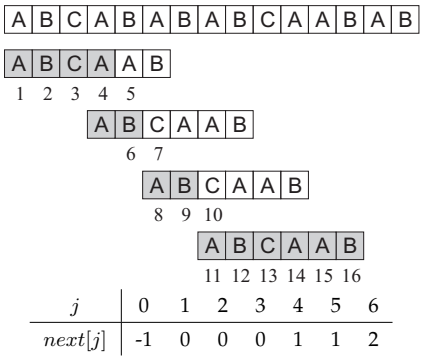
Der KMP-Algorithmus hat eine Zeitkomplexität von O(n) , wo n die Länge des Textes ist, da der Algorithmus jedes Zeichen im Text nur einmal vergleicht. Es ist besonders nützlich, wenn es viele Wiederholungen des Musters im Text gibt.

Im Gegensatz zu anderen Algorithmen wie dem Brute-Force-Algorithmus benötigt der KMP-Algorithmus vorab einen Preprocessing-Schritt, um das failure-array zu berechnen.

Knuth-Morris-Pratt Knuth-Morris-Pratt

- Führt zuerst eine Präfix-Analyse durch und Berechnet eine *next*-Tabelle
- Baut auf der Brute-Force-Suche auf, schiebt das Suchmuster aber in größeren Schritten vorwärts
- Komplexität: $\mathcal{O}(n)$

4.6 Knuth-Morris-Pratt-Algorithmus am Beispiel [45]



j	0	1	2	3	4	5	6
$next[j]$	-1	0	0	0	1	1	2

Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

```java
public int search(String haystack, String needle) {
    int[] lps = computeLPS(needle);
    int i = 0, j = 0;
    while (i < haystack.length()) {
```

```
        if (needle.charAt(j) == haystack.charAt(i)) {
            i++;
            j++;
        }
        if (j == needle.length()) {
            return i - j;
        } else if (i < haystack.length() && needle.charAt(j) != ↩
             haystack.charAt(i)) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i = i + 1;
            }
        }
    }
    return -1;
}
```

4.7 Berechnung der next-Tabelle [46]

j		$next[j]$
1	A B C A A B A B C A A B	0
2	A B C A A B A B C A A B	0
3	A B C A A B A B C A A B	0
4	A B C A A B A B C A A B	1
5	A B C A A B A B C A A B	1

Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

```
public int[] computeLPS(String pattern) {
    int[] lps = new int[pattern.length()];
    int j = 0;
    for (int i = 1; i < pattern.length();) {
        if (pattern.charAt(i) == pattern.charAt(j)) {
            lps[i] = j + 1;
            i++;
            j++;
        } else {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                lps[i] = 0;
                i++;
```

```
            }
        }
    }
    return lps;
}
```

Kapitel 5

Reguläre Ausdrücke

5.1 Zweck [48]

- **Reguläre Ausdrücke** (regular expressions) sind mächtiges Werkzeug — Reguläre Ausdrücke
  - testen, ob ein String ein Muster enthält
  - Teile aus einem String extrahieren
  - Teile eines Strings ersetzen
- Bestehen aus
  - einem **Muster** (pattern), — Muster
  - das auf einen String angewendet wird
- Wenn das Muster im String vorkommen, sagt man es **matched** — matched

Reguläre Ausdrücke können die Arbeit mit Strings und Textdaten erheblich vereinfachen.

Technisch gesehen basieren Reguläre Ausdrücke auf den *Deterministischen Endlichen Automaten (DEA)*, d. h. sie sind eine spezielle Form, einen solchen Automaten zu beschreiben. Hat man einen Regulären Ausdruck erstellt kann man testen, ob ein gegebener String *matched*, d. h. ob der durch ihn beschriebene DEA den String akzeptiert oder nicht. Wenn der Ausdruck matched, kann man die Position im String bestimmen, an der dies erfolgt ist und basierend darauf Teile aus dem String extrahieren oder auch ersetzen,

Ein Regulärer Ausdruck wird durch ein Muster beschrieben, das eine spezielle Syntax besitzt.

5.2 Grundlagen regulärer Ausdrücke [49]

- Alle Zeichen, bis auf `. | ( ) [ ] { } + \ ^ $ * ?` matchen sich selbst
- `.` steht für jedes beliebige Zeichen
- Können mit `Pattern.matches` auf einen String angewandt werden
  - `true`, wenn es einen Match gibt
  - `false`, wenn die RegEx nicht matched



```
Pattern.matches("cat.*", "cat and dog"); // -> true
Pattern.matches("cat.*", "catch"); // -> true
Pattern.matches("cat.*", "Cat"); // -> false
```

Da Reguläre Ausdrücke kein fester Bestandteil der Sprache sind, werden Sie in Java durch Strings repräsentiert, die man mithilfe der Klasse `Pattern` in ein entsprechendes Objekt umwandeln kann.

```
Pattern p = Pattern.compile("cat");
Matcher m = p.matcher("dogs and cats");
if (m.find()) {
    System.out.println(m.start()); // -> 9
    System.out.println(m.end()); // -> 12
}
```

Ein Regulärer Ausdruck ist selbst eine Zeichenkette, in der jedes Zeichen (mit einigen Ausnahmen) zuerst einmal sich selbst matched, z. B. matched `cat` genau den String „cat“. Sobald der Reguläre Ausdruck gefunden wurde (matched), kann als Ergebnis die Position ausgelesen werden. Will man weitere Treffer im String finden, muss man `find()` wiederholt aufrufen.

Neben den Buchstaben, die sich selbst matchen, gibt es eine Reihe von speziellen Zeichen, die z. B. als Wildcard oder Gruppierungszeichen dienen. Diese geben Regulären Ausdrücken ihre Flexibilität. Reguläre Ausdrücke sind relativ komplex, sodass hier nur ein kurzer Überblick gegeben werden kann.

- `.` = ein beliebiges Zeichen
- `|` = oder
- `(`, `)` = umschließt eine Gruppe
- `[`, `]` = umschließt eine Zeichenklasse
- `{ }` = umschließt Wiederholungen
- `*` = 0 oder mehr Vorkommen
- `+` = 1 oder mehr Wiederholungen
- `?` = 0 oder 1 Vorkommen
- `\` = Escape Zeichen
- `^` = Anfang der Zeile
- `$` = Ende der Zeile
- `^` = Negation in einer Zeichenklasse

5.3 Ersetzungen in Strings [50]

`replaceFirst` und `replaceAll` erlauben Ersetzungen in Strings

- `replaceFirst` ersetzt nur das erste Vorkommen — `replaceFirst`
- `replaceAll` ersetzt alle Vorkommen — `replaceAll`

```
String str = "Mathematik ist toll";
String newStr = str.replaceFirst("Mathematik", "Informatik");

System.out.println(newStr); // -> "Informatik ist toll"

System.out.println("leet".replaceFirst("e", "3")); // -> "l3et"
System.out.println("leet".replaceAll("e", "3")); // -> "l33t"
```

Eine häufige Anwendung von Regulären Ausdrücken ist die Ersetzung von Texten in Strings. Hierzu dienen die `replaceFirst`- und `replaceAll`-Methoden von `String`. Sie bekommen als ersten Parameter einen Regulären Ausdruck und als zweiten die gewünschte Ersetzung. Bei `replaceFirst` wird der erste Match im String ersetzt, bei `replaceAll` werden alle ersetzt.

5.4 Matcher-Objekt [51]

Matcher-Objekt

Matcher-Objekt

- `matcher`-Methode von `Pattern` liefert Objekt zurück
- erlaubt den gematchten Teil zu extrahieren

matcher

```
String s = "cats and dogs";
Matcher m = Pattern.compile("at").matcher(s);
m.find();
System.out.printf("%s[%s]%s",
    s.substring(0, m.start()),
    s.substring(m.start(), m.end()),
    s.substring(m.end()));
```

Ausgabe

```
c[at]s and dogs
```

Methode `matcher` ein Objekt vom Typ `Matcher` zurück, das eine Reihe von nützlichen Methoden hat, um auf die Ergebnisse zuzugreifen.

Matcher

- `start()`: Start-Index des Treffers
- `end()`: End-Index des Treffers
- `group(n)`: Zugriff auf den n^{ten} Match (bei Gruppen)

5.5 Anker [52]

Anker (anchors) erlauben es, das Pattern auf den Anfang oder das Ende eines Strings zu beschränken

Anker

- ^: Anfang der Zeile
- $: Ende der Zeile
- \A: Anfang des Strings
- \Z: Ende des Strings
- \b: Wortgrenze
- \B: keine Wortgrenze

Mit Ankern kann man den Regulären Ausdruck auf bestimmte Grenzen einschränken, z. B. auf den Anfang einer Zeile eines Strings etc.

String: `"this is\nthe time"`

- `^the` ⇒ `"this is\n[the] time"`
- `is$` ⇒ `"this [is]\nthe time"`
- `\Athis` ⇒ `"[this] is\nthe time"`
- `\Athe` ⇒ kein Match
- `\bis` ⇒ `"this [is]\nthe time"`
- `\Bis` ⇒ `"th[is] is\nthe time"`

5.6 Zeichenklassen [54]

Zeichenklassen (character classes)

Zeichenklassen

- Zeichen in eckigen Klammern `[aeio]` matchen jedes einzelne Zeichen
- Sonderzeichen (?, + …) haben keine Bedeutung
- Bereiche können angegeben werden (z. B. `[a-z]`)
- Negierung der Zeichen über ^
- Spezielle Abkürzungen
  - \s: Whitespace
  - \S: Alles außer Whitespace
  - \d: Ziffer
  - \D: Alles außer Ziffern

Zeichenklassen können verwendet werden, um bestimmte Zeichen aus einer Liste von Zeichen zu matchen. Sie werden durch eckige Klammern (`[...]`) definiert.

String: `"Der Preis ist 10 EURO"`

- `[aeiou]` ⇒ `"D[e]r Preis ist 10 EURO."`
- `\s` ⇒ `"Der[ ]Preis ist 10 EURO."`

- `[.]` ⇒ `"Der Preis ist 10 EURO[.]"`
- `[A-Z]` ⇒ `"[D]er Preis ist 10 EURO."`
- `[E-Z]` ⇒ `"Der [P]reis ist 10 EURO."`
- `[0-9]` ⇒ `"Der Preis ist [1]0 EURO."`
- `[0-9][0-9]` ⇒ `"Der Preis ist [10] EURO."`
- `[^Daeiou]` ⇒ `"De[r] Preis ist 10 EURO."`

Das Beispiel zeigt die Verwendung von Zeichenklassen in Regulären Ausdrücken und zeigt jeweils den ersten Match.

- `[aeiou]` ⇒ matched ein Zeichen und zwar entweder ein `a`, ein `e`, ein `i` ein `o` oder ein `u`
- `\s` ⇒ matched ein Whitespace, also ein Leerzeichen, ein Tab oder ein Zeilenende
- `[.]` ⇒ matched ein Zeichen und zwar den Punkt. Da der Punkt in einer Zeichenklasse (in `[]`) angegeben ist, hat er nicht mehr die ursprüngliche Bedeutung ein beliebiges Zeichen, sondern ist wirklich nur ein `.`.
- `[A-Z]` ⇒ matched alle Großbuchstaben von `A` bis `Z`.
- `[E-Z]` ⇒ matched alle Großbuchstaben im Bereich `E` bis `Z`.
- `[0-9]` ⇒ matched alle Ziffern von `0` bis `9`
- `[0-9][0-9]` ⇒ matched alle zweistelligen Ziffernfolgen von `00` bis `99`.
- `[^Daeiou]` ⇒ matched alles außer den Zeichen `D`, `a`, `e`, `i`, `o` und `u`.

5.7 Wiederholungen [56]

Manchmal weiß man nicht genau, wie oft ein Zeichen oder Muster vorkommt. Dann kann man eine **Wiederholung** (repetition) nach dem regulären Ausdruck angeben

Wiederholung

- `*`: 0 oder mehr Vorkommen (Vorsicht!) `*`
- `+`: 1 oder mehr Wiederholungen `+`
- `?`: 0 oder 1 Vorkommen (Vorsicht!) `?`
- `{m,n}`: Mindestens m, höchstens n Vorkommen `{m,n}`
- `{m,}`: Mindestens m Vorkommen `{m,}`
- `{,n}`: Höchstens n Vorkommen `{,n}`
- `{m}`: Genau m Vorkommen `{m}`

Über Wiederholungen kann man festlegen, wie oft ein Pattern mindestens oder höchstens vorkommen kann. Bei `*` und `?` ist Vorsicht geboten, weil sie auch kein Vorkommen des Musters als Treffer gilt. So matched z. B. `[0-9]?` jedes Zeichen und nicht wie möglicherweise erwartet nur Zahlen. Ein Muster, dass nur aus einer Zeichenklasse und einem `?` oder `*` besteht ist in den allermeisten Fällen falsch.

`"Der Preis ist 10 EURO"`

- `.?[aeiou]{1}` ⇒ `"[De]r Preis ist 10 EURO."`

- `.?[aeiou]{2,3}` ⇒ `"Der P[rei]s ist 10 EURO."`
- `[0-9]+` ⇒ `"Der Preis ist [10] EURO."`
- `[0-9]?` ⇒ `"[]Der Preis ist 10 EURO."`

5.8 Greedy Match [58]

- Ein regulärer Ausdruck versucht immer so viele Zeichen wie möglich zu matchen (**greedy match**)

greedy match

```
<.*> mit <a href="x">Hallo</a> => <a href="x">Hallo</a>
```

- Will man das nicht, muss man an den Regulären Ausdruck ? anhängen, um einen **non-greedy match** zu bekommen

non-greedy match

```
<.*?> mit <a href="x">Hallo</a> => <a href="x">
```

Ein häufiger Effekt, mit dem Anfängern im Umgang mit regulären Ausdrücken nicht rechnen, ist der *Greedy Match* (greedy = gierig): Ein Regulärer Ausdruck versucht maximal viele Zeichen zu machten, d. h. wenn er ein Wildcard enthält wird nicht beim ersten Treffer des Teils nach dem Wildcard abgebrochen, sondern es wird weitergesucht. Wird weiter hinten im String ein Treffer gefunden, so werden maximal viele Zeichen als gematcht betrachtet. Aus diesem Grund, matched `<.*>` nicht nur den öffnenden HTML-Tag, sondern matched bis zum Ende des schließenden Tags.

Will man, dass ein Regulärer Ausdruck *nicht* greedy matched, muss man an den Wildcard ein Fragezeichen ? anhängen. So matched `<.*?>` tatsächlich nur den öffnen Tag und nicht dessen Inhalt.

5.9 Gruppierungen [59]

- Innerhalb eines Regulären Ausdrucks kann man mit `()` **Gruppen** bilden, die dann als ein einziger Ausdruck betrachtet werden
- Auf die geklammerten Ausdrücke kann dann nach dem Match zugegriffen werden

Gruppen

```
String s = "oh ein Rabababer";
Matcher m = Pattern.compile("(ab)+").matcher(s);
m.find();
System.out.printf(m.group()); // -> "ababab"

s = "Es ist 2022-09-20";
m = Pattern.compile("(2[0-9]{3})-([0-2][0-9])-([0-3][0-9])").matcher(s);
m.find();
System.out.printf(m.group(1)); // -> 2022
```

```
System.out.printf(m.group(2)); // -> 09
System.out.printf(m.group(3)); // -> 20
```

Man kann mithilfe von Klammern `()` im Regulären Ausdruck Gruppen bilden. Diese dienen zwei Zwecken:

1. Angehängte Wildcards wirken dann auf die gesamte Gruppe und nicht mehr nur auf das letzte Zeichen.
2. Man kann die Gruppen in Ersetzungsoperationen (siehe unten) oder beim Auslesen der Matches einzeln ansprechen.

Das Beispiel oben zeigt die Funktionsweise der Gruppen: Die Elemente des Datums können über das `Matcher`-Objekt, das bei der `matcher`-Methode zurückgegeben wird einzeln angesprochen werden.

5.10 Gruppierungen in Ersetzungen [60]

Backreference

Backreference

- Gruppen können auch in Ersetzungen referenziert werden
- Adressierung über `$1`, `$2`, …

```
String s = "Es ist 2022-09-20.";
String p = "(2[0-9]{3})-([0-2][0-9])-([0-3][0-9])";
s.replaceAll(p, "$3.$2.$1");
```

Ausgabe

```
Es ist 20.09.2022.
```

Das Beispiel zeigt eine Ersetzungsoperation mit `replaceAll` bei der über einen Regulären Ausdruck ein Datum vom ISO-Format (JJJJ-MM-TT) in das deutsche Datumsformat (TT.MM.JJJJ) umgewandelt wird. Die einzelnen Teile des Datums werden über Gruppen in ihre Teile zerlegt und dann über die Backreferences `$3`, `$2`, `$1` wieder in die neue Reihenfolge gebracht.

5.11 Weitere Möglichkeiten [61]

Mit Regulären Ausdrücken kann man noch viel mehr machen

- Lookahead/Lookbehind: Umgebung beim Match berücksichtigen
- Schachtelung von Regulären Ausdrücken
- Namen für Rückreferenzen
- Bedingte Gruppen
- Unterroutinen

Reguläre Ausdrücke bieten noch deutlich mehr Möglichkeiten, die hier nicht im Detail beschrieben werden können.

Kapitel 6

Sortieren

6.1 Einleitung [63]

In der Informatik ist das Sortieren von Daten eine wichtige Aufgabe. Es gibt viele verschiedene Algorithmen, die verwendet werden können, um Daten zu sortieren, und jeder hat seine eigenen Stärken und Schwächen. In diesem Kapitel werden wir uns einige der gängigsten Sortieralgorithmen ansehen und ihre Anwendungen sowie ihre Vor- und Nachteile untersuchen. Wir werden auch sehen, wie man bestimmt, welcher Algorithmus für eine bestimmte Aufgabe am besten geeignet ist und wie man ihn implementiert.

Sortierverfahren Sortierverfahren

- Algorithmen, um Daten anhand eines Kriteriums zu ordnen
- Algorithmen unterschiedlicher Komplexität
  - Eingabe: Unsortierte Daten
  - Ausgabe: Daten, die anhand des **Sortierkriteriums** sortiert sind Sortierkriteriums

Ein Verfahren heißt **stabil**, wenn es die relative Reihenfolge gleicher Schlüssel in den Daten beibehält. stabil

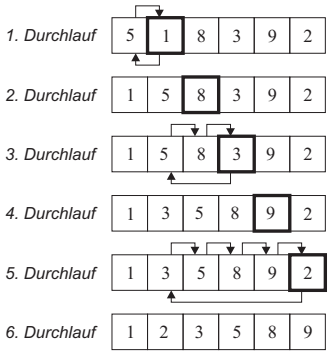
6.2 Insertionsort [64]

Insertionsort ist ein einfacher, stabiler Sortieralgorithmus, der auf der Idee des Sortierens einer Hand von Karten basiert. Der Algorithmus geht durch eine Liste von Elementen und fügt jedes Element an der richtigen Stelle in die Liste ein. Dies geschieht, indem das aktuelle Element mit jedem Element verglichen wird und es an der richtigen Stelle platziert wird. Dieser Vorgang wiederholt sich für jedes Element in der Liste, bis die gesamte Liste sortiert ist. Insertionsort hat eine Zeitkomplexität von $\mathcal{O}(n^2)$ im Worst-Case-Szenario, was es für große Datenmengen ungeeignet macht, es eignet sich jedoch für kleine Datenmengen oder bereits teilweise sortierte Daten. Insertionsort

1. Starte mit der ersten Karte einen neuen Stapel.
2. Nimm jeweils die nächste Karte des Originalstapels und füge diese an der richtigen Stelle in den neuen Stapel ein.

Insertionsort in Pseudocode

```
for i := 1 to n do
  m := F[i]; /* zu merkendes Element */
  j := i;
  while j > 1 do
    if F[j - 1] >= m then
      /* Verschiebe F[j - 1] eine Position nach rechts */
      F[j] := F[j - 1];
      j := j - 1
    else
      Verlasse innere Schleife
    fi
  od;
  F[j] := m /* j zeigt auf Einfügeposition */
od
```



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Insertionsort in Java

```java
public void sort(int[] data) {
    for (int i = 1; i < data.length; i++) {
        int m = data[i];
        int j = i;
        while (j > 0) {
            if (data[j - 1] >= m) {
                /* Verschiebe data[j - 1] eine Position nach rechts */
                data[j] = data[j - 1];
                j--;
            }
            else {
                break;
            }
        }
        data[j] = m;
    }
}
```

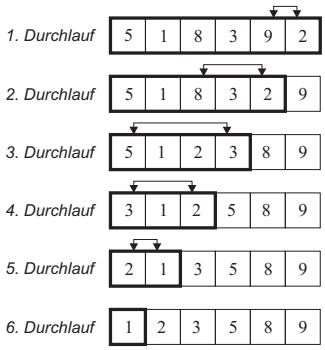
6.3 Selectionsort [68]

Selectionsort

Selectionsort ist ein einfacher, in-place Sortieralgorithmus, bei dem das größte Element in einer Liste ausgewählt und an die letzte Stelle platziert wird. Dann wird der Prozess für die restlichen Elemente wiederholt, indem das nächstgrößere Element ausgewählt und an die nächste Stelle platziert wird, und so weiter, bis die Liste vollständig sortiert ist. Der Algorithmus durchläuft die Liste mehrere Male und vergleicht jedes Element mit jedem anderen Element, um das größte Element zu finden und es an die richtige Stelle zu verschieben. Selectionsort hat eine Zeitkomplexität von $\mathcal{O}(n^2)$ im Worst-Case-Szenario, was es für große Datenmengen ungeeignet macht, es eignet sich jedoch für kleine Datenmengen oder bereits teilweise sortierte Daten.

1. Suche die größte Karte im Stapel.
2. Stecke die größte Karte ans Ende des Stapels.
3. Wiederhole die Schritte für die verbleibenden Karten.

```
p := n;
while p > 0 do
  g := Index des grösten Elementes aus F im Bereich 1 ... p;
  Vertausche Werte von F[p] und F[g];
  p := p - 1
od
```



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

```java
public void sort(int[] data) {
    int pos = data.length - 1;
    while (pos >= 0) {
        // bestimme größtes Element
        int indexMax = 0;
        for (int i = 0; i <= pos; i++) {
            if (data[i] > data[indexMax]) {
                indexMax = i;
            }
        }
        // tausche array[pos] mit diesem Element
```

```
        swap(data, pos, indexMax);
        pos--;
    }
}
```

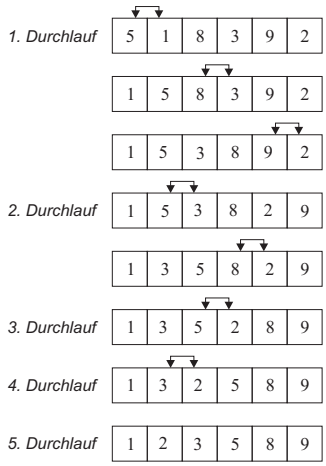
6.4 Bubblesort [72]

Bubblesort ist ein einfacher, aber ineffizienter Sortieralgorithmus. Der Algorithmus geht durch eine Liste von Elementen und vergleicht jedes Element mit seinem Nachbar. Wenn ein Element größer ist als sein Nachbar, werden die beiden Elemente vertauscht. Dieser Vorgang wird wiederholt, bis keine weiteren Vertauschungen mehr notwendig sind, was bedeutet, dass die Liste sortiert ist. Bubblesort hat eine Zeitkomplexität von $\mathcal{O}(n^2)$ im Worst-Case-Szenario und $\mathcal{O}(n)$ im best case, was es für große Datenmengen ungeeignet macht.

Bubblesort

1. Durchlaufe die Liste und vergleiche die Karte mit dem rechten Nachbarn.
2. Wenn der rechte Nachbar kleiner ist, vertausche die Karten.
3. Wiederhole die Schritte, bis keine Vertauschung mehr auftritt.

```
do
  for i := 1 to n - 1 do
    if F[i]> F[i + 1] then
      Vertausche Werte von F[i] und F[i + 1]
    fi
  od
until keine Vertauschung mehr aufgetreten
```



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

```
public void sort(int[] data) {
    boolean swapped;
```

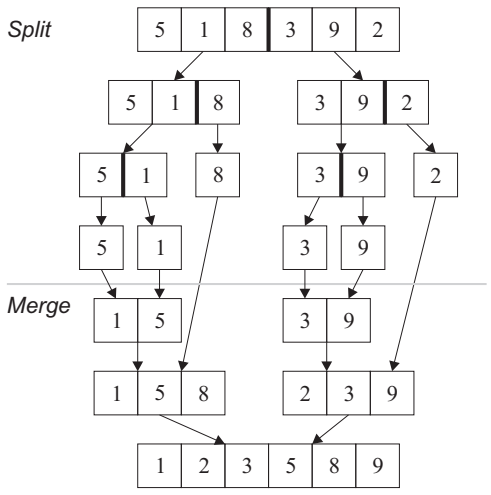
```
    do {
        swapped = false;
        for (int i = 0; i < data.length - 1; i++) {
            if (data[i] > data[i + 1]) {
                // Elemente vertauschen
                swap(data, i, i + 1);
                swapped = true;
            }
        }
    } while (swapped);
}
```

6.5 Mergesort [76]

Mergesort

Mergesort ist ein effizienter, stabiler Sortieralgorithmus, der auf dem Prinzip des „Teilen und Herrschens“ basiert. Der Algorithmus teilt die Liste in zwei gleich große Teile und sortiert jeden Teil rekursiv. Sobald beide Teile sortiert sind, werden sie zusammengefügt (merge), indem die Elemente aus beiden Teilen verglichen und in eine neue sortierte Liste eingefügt werden. Dieser Vorgang wiederholt sich, bis die gesamte Liste sortiert ist. Mergesort hat eine Zeitkomplexität von $\mathcal{O}(n \log n)$ im Worst-Case-Szenario, was es zu einem sehr effizienten Algorithmus für große Datenmengen macht. Es benötigt jedoch zusätzlichen Speicherplatz, um die zwischengespeicherten Teillisten zu halten.

1. Halbiere den Kartenstapel rekursiv in kleinere Teillisten (z. B. 1, 2 oder drei Karten).
2. Sortiere die Teillisten einzeln
   (bei einem Element pro Liste entfällt dieser Schritt).
3. Füge die Teillisten wieder schrittweise zusammen.
   - Vergleiche die ersten Elemente der Teillisten und übernimm das kleinere.
   - Entferne das Element aus der Teilliste und wiederhole den Vergleich.



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

```
void msort(int[] array, int le, int ri, int[] helper) {
    int i, j, k;
    if (ri > le) {
        // zu sortierendes Feld teilen
        int mid = (ri + le) / 2;

        // Teilfelder sortieren
        msort(array, le, mid, helper);
        msort(array, mid + 1, ri, helper);

        // Hilfsfeld aufbauen
        for (k = le; k <= mid; k++) {
          helper[k] = array[k];
        }

        for (k = mid; k < ri; k++) {
          helper[ri + mid - k] = array[k + 1];
        }
```

```
        // Ergebnisse mischen über Hilfsfeld
        i = le; j = ri;
        for (k = le; k <= ri; k++) {
            if (helper[i] < helper[j]) {
                array[k] = helper[i++];
            } else {
                array[k] = helper[j--];
            }
        }
    }
}

public void sort(int[] data) {
    int[] helper = new int[data.length];
    msort(data, 0, data.length - 1, helper);
}
```

6.6 Quicksort [80]

Quicksort
Pivot

Quicksort ist ein schneller, in-place Sortieralgorithmus, der auf der Idee des „Teilen und Herrschens“ basiert. Der Algorithmus wählt ein Element als **Pivot** und teilt die Liste in zwei Teile: Elemente, die kleiner als das Pivot-Element sind und Elemente, die größer sind. Dann wendet er sich rekursiv an jeden Teil, um diese Teile sortiert zu machen. Dieser Vorgang wiederholt sich, bis die gesamte Liste sortiert ist. Quicksort hat eine durchschnittliche Zeitkomplexität von $\mathcal{O}(n \log n)$ was es zu einem der schnellsten Sortieralgorithmen für große Datenmengen macht. Es hat jedoch auch eine worst-case-Zeitkomplexität von $\mathcal{O}(n^2)$, die durch die Wahl des

falschen Pivot-Elements verursacht werden kann. Um dies zu vermeiden, kann man diverse Pivot-Auswahl-Methoden verwenden, um die bestmögliche Leistung zu erreichen.

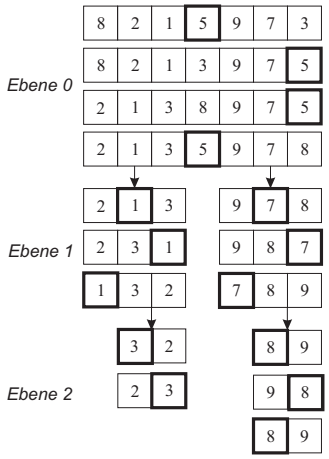
1. Wähle ein *Pivotelement*.
2. Spalte die die Karten in zwei Teillisten:
   - Kleiner (oder gleich) dem Pivotelement.
   - Größer (oder gleich) dem Pivotelement.
3. Wiederhole die Schritte 1. und 2. bis die Teillisten die Länge 1 haben.
4. Die Gesamtliste ist sortiert.

```
Quicksort (F, u, o)
Eingabe: eine zu sortierende Folge F,
die untere und obere Grenze u, o

if o > u then
  Bestimme Position p des Pivot-Elementes;
  pn := Zerlege(F, u, o, p);
  Quicksort (F, u, pn - 1);
  Quicksort (F, pn + 1, o)
fi
```

```
Zerlege (F, u, o, p) -> pn
Eingabe: zu zerlegende Folge F, untere und obere Grenze u, o,
Position p des Pivot-Elementes
Ausgabe: neue Position pn des Pivot-Elementes

pn := u;
pv := F[p];
Tausche F[p] und F[o]; // Pivot-Element nach rechts
for i := u to o - 1 do
  if F[i] <= pv then
    Tausche F[pn] und F[i];
    pn := pn + 1;
  fi
od
Tausche F[o] und F[pn]; // Pivot-Element zurück
return pn
```



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

```
int partition(int[] array, int u, int o, int p) {
    int pn = u;
    int pv = array[p];
    // Pivot-Element an das Ende verschieben
    swap(array, p, o);
    for (int i = u; i < o; i++) {
        if (array[i] <= pv) {
            swap(array, pn++, i);
        }
    }

    // Pivot-Element an die richtige Position kopieren
    swap(array, o, pn);

    // neue Pivot-Position zurückgeben
    return pn;
}
```

```
void qsort(int[] array, int u, int o) {

    // Pivot-Element bestimmen
    int p = (u + o) / 2;
    if (o > u) {
        // Feld zerlegen
        int pn = partition(array, u, o, p);
        // und Partitionen sortieren
        qsort(array, u, pn - 1);
        qsort(array, pn + 1, o);
    }
}
```

```
public void sort(int[] data) {
    qsort(data, 0, data.length - 1);
}
```

6.7 Verfahren im Vergleich [86]

Verfahren	Stabilität	Vergleiche (im Mittel)
Selectionsort	instabil	$1/2\ n^2$
Insertionsort	stabil	$1/4\ n^2$
Bubblesort	stabil	$1/2\ n^2$
Mergesort	stabil	$n\ \log_2(n)$
Quicksort	instabil	$2n\ \ln(n)$

Index

Θ-Notation, 9

Algorithmen, 1
Anker, 26
average case, 5

Backreference, 29
best case, 5
Binäre Suche, 12
Boyer-Moore-Algorithmus, 17, 18
Brute-Force-Algorithmus, 16
Bubblesort, 34

determiniert, 2
deterministisch, 2

Fisher-Yates-Verfahren, 14

greedy match, 28
Gruppen, 28

Insertionsort, 31

Knuth-Morris-Pratt, 19, 20
Knuth-Shuffle, 14
Komplexitätstheorie, 4

Landau-Notation, 7

matched, 23
Matcher-Objekt, 25
Mergesort, 35
Mischverfahren, 13
Muster, 23

non-greedy match, 28

O-Notation, 7
obere Schranke, 7

Pivot, 36
Prozess, 1

Quicksort, 36

Reguläre Ausdrücke, 23

Selectionsort, 33
Sequentielle Suche, 11
Sortierkriteriums, 31
Sortierverfahren, 31
Speicherkomplexität, 5
stabil, 31

terminierend, 2

Wiederholung, 27
worst case, 5

Zeichenklassen, 26
Zeitkomplexität, 5

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Object und Wrapper-Klassen



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**
- 1.1 Fragen [5] . . . 1
- 1.2 Begriffe [6] . . . 1
- 1.3 Links zur Java-Insel [7] . . . 1

2 Object **2**
- 2.1 Die Klasse Object [9] . . . 2
- 2.2 hashCode() [10] . . . 2
- 2.3 equals() und == [11] . . . 3
- 2.4 Beispiel: equals() und hashCode() [12] . . . 3
- 2.5 Beispiel: equals() und == [15] . . . 5
- 2.6 Achtung: Falle bei String [17] . . . 5
- 2.7 Praxistipp: Test auf null vermeiden [19] . . . 6
- 2.8 toString() [20] . . . 7
- 2.9 clone() [22] . . . 7
- 2.10 Die Hilfsklasse java.util.Objects [25] . . . 9

3 Wrapper-Klassen **11**
- 3.1 Wrapper-Klassen [28] . . . 11
- 3.2 Beispiel: Wrapper [30] . . . 12
- 3.3 Autoboxing [31] . . . 12
- 3.4 Praxistipp: Autoboxing und == [32] . . . 13

4 Optional **14**
- 4.1 Motivation [36] . . . 14
- 4.2 Optional [37] . . . 14
- 4.3 Beispielhafte Verwendung [39] . . . 15
- 4.4 Primitive Datentypen [40] . . . 16

5 Quiz **18**
- 5.1 Quiz: Equals I [44] . . . 18
- 5.2 Quiz: Equals II [46] . . . 19
- 5.3 Quiz: Vergleich [48] . . . 19

Index **i**

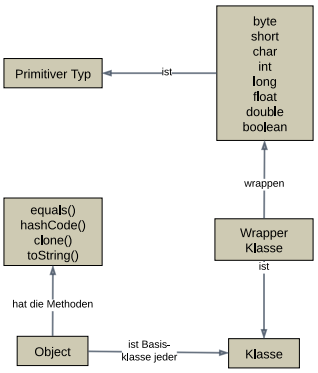
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie sorge ich dafür, dass alle Objekte gemeinsame Methoden haben?
- Wie hole ich die primitiven Datentypen ins objektorientierte Boot?
- Wie werde ich die `NullpointerException` los?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 11.1 Object ist die Mutter aller Klassen
- 11.3 Die Utility-Klasse java.util.Objects
- 11.5 Wrapper-Klassen und Autoboxing
- 13.6 Optional ist keine Nullnummer

Kapitel 2

Object

2.1 Die Klasse Object [9]

Object

- Die Klasse `Object` ist die Wurzel aller Vererbungshierarchien
- Jede Java-Klasse erbt direkt oder indirekt von `Object`
- Wenn eine Klasse von keiner anderen erbt, erzeugt der Compiler automatisch ein `extends Object`, d. h. aus
  `public class Klasse {}` wird
  `public class Klasse extends Object {}`
- `Object` enthält einige Methoden, die es an alle anderen Klassen vererbt
  - `equals()`
  - `hashCode()`
  - `toString()`
  - `clone()`

Die anderen Methoden, die von der Klasse `Object` zur Verfügung gestellt werden, werden wir später noch im Rahmen anderer Themen wie Threads oder Reflection diskutieren.

2.2 hashCode() [10]

hashCode()

- `hashCode()` dient dazu, einen möglichst eindeutigen Hash-Wert für das Objekt zu erzeugen
- Die Implementierung von `Object` verwendet einen „einmaligen“ Schlüssel für den Hash-Wert, unabhängig von den Daten im Objekt
- Man überschreibt `hashCode()`, um einen Hash-Wert aus den Daten des Objekts zu berechnen

Wenn man equals() überschreibt, so muss man auch hashCode() überschreiben und umgekehrt

Möglichst eindeutig bedeutet, dass es vorkommen kann, dass zwei Objekte mit unterschiedlichem Inhalt denselben Hash-Wert haben. Dies ist nicht zu vermeiden, da der Hash-Wert nur ein einfacher 32bit `int` Wert ist und daher nicht immer eindeutig sein kann. Man spricht in dem Fall, dass zwei unterschiedliche Objekte denselben Hash-Wert haben von einer **Hash-Kollision** (hash collision) oder einer Kollision der Hash-Werte. Solche Kollisionen führen nicht zu Fehlverhalten des Programms, sind aber trotzdem aus Gründen der Effizienz zu minimieren. Um sie zu vermeiden, muss man versuchen, die erzeugten Hash-Wert möglichst breit über den verfügbaren Wertebereich eines 32bit Integers zu streuen und dafür zu sorgen, dass ähnliche Daten nicht unbedingt ähnliche Hash-Werte bekommen.

Hash-Kollision

Verwendet man die *Standard-Implementierung* von `hashCode()` aus `Object` (d. h. wenn man `hashCode()` nicht überschrieben hat), so haben zwei unterschiedliche Objekte fast niemals denselben Hash-Code, auch wenn ihr Inhalt vollkommen gleich ist. Zwei unterschiedliche Objekte können auch hier denselben Hashcode haben, falls es zu einer Kollision kommt. Da die ursprüngliche Implementierung von `Object` beim Überschreiben der `hashCode`-Methode verloren geht, kann man sich zu jedem Objekt später über die Methode `System.identityHashCode(Object o)` diesen ursprünglichen Wert beschaffen.

2.3 equals() und == [11]

- Der == Operator bestimmt, ob zwei Objekt-Referenzen identisch sind (auf dasselbe Objekt zeigen)
- `equals()` stellt fest, ob zwei Objekte gleich sind (den gleichen Inhalt haben)
- Wenn man `equals()` nicht überschreibt, verwendet die Implementierung aus Object den == Operator

==

equals()

Wenn man equals() überschreibt, so muss man auch hashCode() überschreiben und umgekehrt

2.4 Beispiel: equals() und hashCode() [12]

gegebene Klasse

```
public class Mitarbeiter {
    public String name;
    public double gehalt;
    public Date geburtsdatum;
}
```

equals() Methode zur Klasse Mitarbeiter

```
@Override
public boolean equals(Object o) {

    if (o == null) {
```

```
        return false;
    } else if (o == this) {
        return true;
    } else if (o.getClass() == Mitarbeiter.class) {
        Mitarbeiter mi = (Mitarbeiter) o;

        return mi.name.equals(name)
               && mi.gehalt == gehalt
               && mi.geburtsdatum.equals(geburtsdatum);
    } else {
        return false;
    }
}
```

hashCode() Methode zur Klasse Mitarbeiter



```
@Override
public int hashCode() {
    return name.hashCode()
           ^ (int) Double.doubleToLongBits(gehalt)
           ^ geburtsdatum.hashCode();
}
```

Die hier vorgestellte Implementierung ist noch in einigen Punkten naiv und daher verbesserungswürdig. Für eine bessere Implementierung fehlt uns an dieser Stelle aber noch einiges an Theorie.

- Keine der Methoden behandelt null-Werte korrekt
- Die Verwendung von XOR (^) für die Hash-Werte funktioniert zwar, gewährleistet aber noch keine optimale Streuung der Werte

Die Konstruktion `o.getClass()== Mitarbeiter.class` ist uns noch unbekannt und prüft, ob das übergebene Objekt von *genau derselben* Klasse ist, wie das vorhandene. Hierbei wird das sogenannte **Class-Objekt** verwendet, das die Klasse eines Objektes selbst wieder als Objekt repräsentiert. Es gibt pro Klasse immer nur ein einziges Class-Objekt, sodass man hier sicher mit == vergleichen kann. Anstatt `o.getClass()== Mitarbeiter.class` hätte man auch `o.getClass()== this.getClass()` schreiben können.

Class-Objekt

Alternativ könnte man `o instanceof Mitarbeiter` schreiben. Der Unterschied wäre dann aber, dass `equals()` dann `true` liefern könnte, wenn das Objekt mit dem verglichen wird, von einer Subklasse von Mitarbeiter stammt, was häufig nicht das gewünschte Verhalten ist.

Beachten Sie weiterhin, dass die equals-Methode einen Parameter vom Typ `Object` erhält. Wenn man dies ändert, z. B. in dem man im Beispiel Mitarbeiter als Typ verwendet, kommt es zu subtilen Fehlern, weil man hierdurch die Methode `equals()` nicht überschrieben, sondern überladen hat. Man hat also zwei verschiedene equals-Methoden, eine von `Object` geerbt und eine weitere in der Klasse Mitarbeiter hinzugefügt. Der Compiler wird anhand des statischen Typs der Variablen eine der beiden Methoden auswählen, sodass in manchen Fällen die eine

und in manchen Fällen die andere Methode benutzt wird. Auch hier könnte man sich durch Verwendung der `@Overrides`-Annotation vor diesem Fehler selbst schützen.

Eine bessere Implementierung von `hashCode()` (wie sie z. B. von Eclipse generiert würde) sähe wie folgt aus:

```
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result
           + ((geburtsdatum == null) ? 0 : geburtsdatum.hashCode());
    long temp;
    temp = Double.doubleToLongBits(gehalt);
    result = prime * result + (int) (temp ^ (temp >>> 32));
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
}
```

2.5 Beispiel: equals() und == [15]

```
MeinDatum d1 = new MeinDatum(5, 12, 2009);
MeinDatum d2 = new MeinDatum(5, 12, 2009);
MeinDatum d3 = new MeinDatum(3, 7, 1970);

System.out.println(d1 == d2);
System.out.println(d1.equals(d2));
System.out.println(d1.equals(d3));

d2 = d1;

System.out.println(d1 == d2);
```

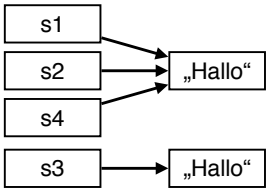
```
false
true
false
true
```

2.6 Achtung: Falle bei String [17]

- Die Java VM optimiert String-Literale und sorgt dafür, dass sie nur ein einziges Mal im Speicher liegen
- Hierzu verwendet sie die Methode `intern()` der Klasse `String`

```
String s1 = "Hallo";
String s2 = "Hallo";
String s3 = new String("Hallo");
String s4 = s3.intern();

System.out.println(s1 == s2); // -> true
System.out.println(s1 == s3); // -> false
System.out.println(s1 == s4); // -> true
```



globale String-Literal-Optimierung

Die VM führt eine so genannte **globale String-Literal-Optimierung** durch, um Speicher zu sparen: Immer wenn ein String-Literal verwendet wird, schaut sie zuerst nach, ob dieses schon irgendwo im Speicher liegt und falls dem so ist, gibt sie eine Referenz auf das bereits existierende Objekt zurück, statt ein neues anzulegen. Dies Optimierung ist möglich, da es sich bei String um ein immutable Objekt handelt, d. h. ein Objekt, das nach seiner Erzeugung nicht mehr verändert werden kann. Daher ist es für zwei Verwender nicht relevant, ob sie zwei verschiedene Objekte oder ein und dasselbe Objekt verwenden.

Das Anlegen eines Strings mit `new String("LITERAL")` verschwendet Speicher und kostet Performance. Es wurde hier nur gewählt, um die Auswirkungen den String-Literal-Optimierung zu illustrieren.

Weiterhin ist zu beachten, dass durch die hier beschriebene Optimierung Code, der fälschlicherweise == statt `equals()` verwendet, zufälligerweise funktionieren kann, nämlich genau dann, wenn es sich bei den beiden zu vergleichenden Objekten zufälligerweise um Literale handelt.

Es gibt einige Fälle, wo es sich lohnen kann, Strings explizit mit `intern()` zu behandeln und dann mit == zu arbeiten. Dies ist immer dann vorteilhaft, wenn man einen String mit einer sehr großen Anzahl von Literalen vergleichen muss. Im Allgemeinen sollte man aber `intern()` nur sehr selten verwenden.

2.7 Praxistipp: Test auf null vermeiden [19]

Bei Vergleichen von Strings mit String-Literalen kann man sich den Test auf `null` sparen, wenn man die Bedingung umdreht

statt

```
if ((s != null)&& (s.equals("text"))){ ... }
```

schreibt man

```
if ("text".equals(s)){ ... }
```

Dieser Trick funktioniert, weil ein String-Literal selbst ein Objekt ist, auf dem man Methoden aufrufen kann und es nie null sein kann. Der Test auf null ist ohnehin bereits in der `equals()`-Methode enthalten, sodass man ihn sich hier sparen kann.

2.8 toString() [20]

Die `toString()`-Methode

`toString()`

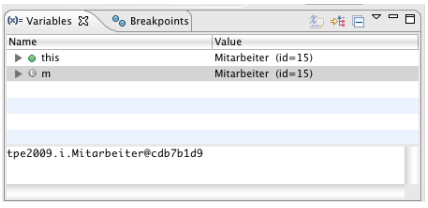
- liefert eine Darstellung des Objekts als String zurück
- wird automatisch bei String-Verknüpfungen aufgerufen
- soll primär beim Debuggen helfen
- dient nicht der Serialisierung von Objekten

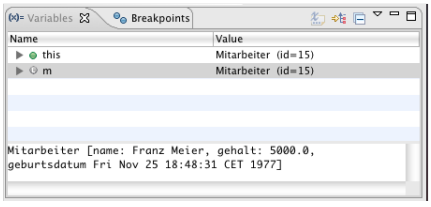
Standardimplementierung gibt den Typ und den Hash-Code aus

```
System.out.println(m + " hat ein Gehalt von " + gehalt);
System.out.println(m.toString() +
       " hat ein Gehalt von " +
       Double.toString(gehalt));
```

Das folgende Beispiel zeigt, wie einem die `toString`-Methode hilfreiche Informationen beim Debuggen eines Programms zur Verfügung stellen kann.

```
@Override
public String toString() {
    return "Mitarbeiter [name: " + name + ", "
           + "gehalt: " + gehalt + ", "
           + "geburtsdatum " + geburtsdatum + "]";
}
```





2.9 clone() [22]

- `clone()` dient dem Kopieren von Objekten
- Erzeugt eine **flache Kopie** (shallow copy) des Objekts

`clone()`

flache Kopie

- Konstruktoren werden nicht aufgerufen
- Anwendung
  - Die protected `clone()`-Methode muss überschrieben werden
  - Klasse muss `Cloneable` implementieren
  - In der ersten Zeile muss `super.clone()` aufgerufen werden
- **Tiefe Kopien** (deep copy) kann man durch rekursiven Aufruf von `clone()` auf den Instanzvariablen erzeugen

Tiefe Kopien

`clone()` hat den Vorteil, dass man mit der Methode einfach Kopien von Objekten herstellen kann. Allerdings hat `clone()` auch einige Nachteile, die es häufig sinnvoll erscheinen lassen auf diesen Ansatz zu verzichten und lieber mit einem Copy-Konstruktor zu arbeiten.

Das Interface Cloneable (zu Interfaces kommen wir später noch) enthält selbst keine Methoden und dient nur dazu die Zustimmung der Klasse dazu zu signalisieren, gecloned zu werden. Man spricht hier von einem **tagging Interface**.

tagging Interface

Da es sich bei `clone()` um eine protected Methode handelt, kann man es nicht zusammen mit Interfaces oder Variablen vom Type einer Superklasse verwenden, die `clone()` noch nicht implementiert hat. Auch ein Cast auf `Cloneable` funktioniert nicht, da `Cloneable` selbst die `clone()`-Methode nicht enthält. Dies ist einer der größten Nachteile von `clone()` und kann als Designfehler angesehen werden.

Das folgende Beispiel zeigt eine beispielhafte Implementierung und Verwendung von `clone()`.

```
public class Mitarbeiter implements Cloneable {
    public String name;
    public double gehalt;
    public Date geburtsdatum;

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Verwendung von clone

```
Mitarbeiter m1 = new Mitarbeiter("Heinz", 5000.00, null);
Mitarbeiter m2 = (Mitarbeiter) m1.clone();

System.out.println(m1 == m2); // -> false
System.out.println(m2.name); // -> Heinz
System.out.println(m2.gehalt); // -> 5000.00
```

Das Werfen der `CloneNotSupportedException` in diesem Beispiel sollte eigentlich vermieden werden, da die Klasse durch implementieren des Cloneable Interfaces explizit zugestimmt hat gecloned zu werden und den Verwendern nur unnötig das Fangen einer Exception aufbürdet, die sowieso nicht geworfen werden wird. Da wir aber hier noch keine Ausnahmen eingeführt haben und damit noch kein try/catch kennen wurde aus didaktischen Gründen die Exception in der Signatur der Methode beibehalten. Korrekterweise würde man das Beispiel normalerweise wie folgt programmieren:

```
public class Mitarbeiter implements Cloneable {
    public String name;
    public double gehalt;
    public Date geburtsdatum;

    public Mitarbeiter(String name, double gehalt, Date geburtsdatum) {
        this.name = name;
        this.gehalt = gehalt;
        this.geburtsdatum = geburtsdatum;
    }

    protected Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError("Cannot happen");
        }
    }
}
```

2.10 Die Hilfsklasse java.util.Objects [25]

Die Klasse `java.util.Objects` in Java bietet einige statische Methoden, die in verschiedenen Anwendungsfällen nützlich sein können.

`java.util.Objects` bietet Hilfsfunktionen

- `equals(Object a, Object b)`
  prüft auf Gleichheit unter Berücksichtigung von `null`
- `hash(Object... vals)`
  Berechnet einen Hashcode für mehrere Werte
- `hashCode(Object o)`
  Berechnet den Hashcode für ein Objekt, das `null` sein könnte
- `requireNonNull(Object o, ...)`
  Prüft, ob ein Objekt `null` ist und wirft eine Ausnahme
- `requireNonNullElseGet(Object o, Object def)`
  Prüft, ob das Objekt `null` ist und gibt das Default-Objekt `def` zurück

- `checkIndex(int index, int length)`, `checkFromToIndex(...)`
  Prüft Indices auf korrekte Grenzen

Die Signaturen der Methoden sind in der wirklichen Klasse anders, als hier angegeben. So sind z. B. die Methoden `requireNonNull` etc. generisch mit einem Typ-Argument. Da generische Typen aber hier noch nicht eingeführt sind, ist die Signatur vereinfacht wiedergegeben.

Kapitel 3

Wrapper-Klassen

3.1 Wrapper-Klassen [28]

- Primitive Typen sind *keine* Objekte
- An manchen Stellen muss man aber Objekte anstatt primitiver Typen verwenden
- **Wrapper-Klassen** (wrapper classes) stellen für jeden primitiven Typ einen passenden Klassen-Typ zur Verfügung

Wrapper-Klassen

Primitiver Typ	Wrapper
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Die Wrapper (Einwickler) haben ihren Namen von der Tatsache, dass sie sich um die primitiven Typen herum legen und sie als Objekt umschließen. Man beachte die Asymmetrie bei `int`/`Integer` und `char`/`Character`: Normalerweise entspricht der Name des Wrappers dem Namen des primitiven Typs nur mit Großbuchstaben. Bei `int` und `char` ist dem aber nicht so.

Alle Wrapper enthalten Konvertierungsmethoden in beide Richtungen, d. h. vom primitiven Typen zum Objekt und vom Objekt zum primitiven Typen.

Weiterhin finden sich in den Wrappern wichtige Konstanten, z. B. `Integer.MAX_INT` als maximaler Wert, den ein `int` annehmen kann.

Alle Wrapper sind als unveränderliche Objekte (immutable Objects) konstruiert. Es kann daher aus Performance-Gründen sinnvoll sein, ein und dasselbe Wrapper-Objekt mehrfach zu verwenden. Bei `Boolean` sollte man normalerweise überhaupt kein Wrapper-Objekt neu anlegen, sondern einfach die beiden vordefinierten Objekte `Boolean.TRUE` und `Boolean.FALSE` verwenden.

3.2 Beispiel: Wrapper [30]

```
int i = 5;
Integer i1 = new Integer(i);
int j = i1.intValue();

Integer i2 = Integer.valueOf("42");
int i3 = Integer.parseInt("42");
```

Das Beispiel zeigt das Anlegen eines `int`-Wrappers über den Konstruktor von `Integer` und das Auslesen des Wertes aus dem Wrapper mit der Methode `intValue()`. Zusätzlich sind zwei Methoden gezeigt, um aus einem String einen `int`-Wert bzw. ein `Integer`-Objekt zu erzeugen. Die Verwendung des Konstruktors ist inzwischen nicht mehr empfohlen, sondern man sollte sich auf das Autoboxing stützen.

3.3 Autoboxing [31]

Seit Java 5 kann der Compiler automatisch zwischen primitiven Typen und Klassen-Typen konvertieren, man spricht hier von **Autoboxing**

Autoboxing

```
Integer k = 5;
int j = k;

Object o = true;
boolean b = new Boolean(true);

Long l = 0L;

for (int i = 0; i < Integer.MAX_VALUE; i++) {
    l += i;
}
```

Der Java-Compiler führt das Autoboxing in beiden Richtungen durch, d. h. es gibt sowohl ein autoboxing als auch ein auto-unboxing.

Die untere Schleife ist mit einem erheblichen Performanceproblem belastet.

Dadurch, dass man statt `long` `Long` geschrieben hat, findet bei jedem Schleifendurchlauf ein Autoboxing statt. Statt also einfach eine `long`-Variable zu erhöhen, werden hier mehrere Milliarden `Long`-Objekte erzeugt und wieder zerstört. Die Geschwindigkeit ist daher um Größenordnungen geringer, ohne dass man dies am Source-Code sehen könnte (bis auf das große „L“).

3.4 Praxistipp: Autoboxing und == [32]

- Durch Autoboxing werden primitiven Typen zu Objekten. Damit gelten aber die Regeln für `equals()` und ==
- Ein Vergleich von Wrapper-Typen mit == ist *immer* falsch

```
public static boolean compare(Integer i1, Integer i2) {
    return i1 == i2;
}

public static void main(String[] args) {
    int i1 = 10;
    int i2 = 10;
    System.out.println(i1 + " == " + i2 + " : " + compare(i1, i2));

    i1 = 2000;
    i2 = 2000;
    System.out.println(i1 + " == " + i2 + " : " + compare(i1, i2));
  }
```

```
Ausgabe
10 == 10 : true
2000 == 2000 : false
```

Die Ausgabe des Programms ist erstaunlich. Beim Wert 2000 sieht man deutlich, dass durch das Autoboxing der `int`-Variablen zwei unterschiedliche Objekte entstanden sind, die nicht damit nicht identisch sind. Daher liefert der ==-Vergleich `false` zurück.

Aber warum passiert das nicht bei dem Wert 10?

Die Klasse `Integer` nimmt eine Optimierung vor: für alle Werte kleiner 128 werden keine neuen `Integer`-Objekte erzeugt, sondern aus einem Cache einfach immer wieder dieselbe Objektreferenz zurückgeliefert. Dies ist möglich, da Integer unveränderlich (immutable) ist und daher für den Verwender nicht unterscheidbar ist, ob er zwei Referenzen auf zwei Objekte oder zwei Referenzen auf dasselbe Objekt benutzt (zumindest so lange er nicht mit == explizit darauf testet). Das hier von Integer angewandte Entwurfsmuster ist das sogenannte Fliegengewicht) (flyweight).

Kapitel 4

Optional

4.1 Motivation [36]

Die Tatsache, dass eine Referenz in Java auf ein Objekt zeigen kann *oder* `null` sein, macht die Programmierung für Anfänger schwierig. Aber auch Profis kämpfen mit der `null`-Referenz, die eine Quelle vieler Fehler sein kann. Der Erfinder dieser Referenz nennt sie rückblickend sein „billion-dollar mistake“.

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...] I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
>
> Tony Hoare, 2009

Andere Programmiersprachen, z. B. Ruby, haben den Fehler nicht gemacht und kennen keine `null`-Referenz. In Ruby gibt es zwar `nil`, dies ist aber ein normales Objekt, mit Methoden und Eigenschaften.

4.2 Optional [37]

`Optional<T>`

In Java kann der Datentyp `Optional<T>` verwendet werden, um auszudrücken, dass ein Wert möglicherweise leer sein kann. Im Gegensatz zu null als leerem Wert, ist `Optional` eine explizite Möglichkeit, um auszudrücken, dass ein Wert nicht vorhanden sein kann und ermöglicht dadurch eine klarere und sicherere Programmierung.

`java.util.Optional` bietet einen sicheren Ersatz für `null`

- `Optional.of(...)`
  - erzeugt ein neues Optional mit einem Wert
  - ist der Wert `null` wird eine `NullpointerException` geworfen
- `Optional.ofNullable(...)` erzeugt ein neues `Optional`

    - wenn das übergebene Objekt `null` ist, wird ein `Optional.empty()` zurückgegeben
    - wenn das übergebene Objekt nicht `null` ist, wird ein `Optional.of(...)` zurückgegeben
- `Optional.empty()` erzeugt ein neues Optional, das leer ist
- `isPresent()` prüft, ob ein Wert vorhanden ist
- `isEmpty()` zeigt an, dass kein Wert vorhanden ist
- `get()`
    - liest den Wert aus dem `Optional` aus
    - ist kein Wert vorhanden, gibt es eine Ausnahme vom Typ `NoSuchElementException`
- `orElse(...)` führt ein `get()`
    - wenn kein Wert vorhanden ist, wird der übergebene Wert geliefert
    - wenn ein Wert vorhanden ist, wird dieser zurückgegeben

Optional kann verwendet werden, um die Fehleranfälligkeit in Java-Programmen zu reduzieren, indem es den Programmierer:innen ermöglicht, Fehler zu behandeln oder auszulösen, wenn ein Wert nicht vorhanden ist.

Ein weiteres Beispiel ist die Rückgabe von Werten aus einer Methode, wo ein Ergebnis nicht immer garantiert werden kann. Anstatt `null` zurückzugeben, könnte man eine Optional-Instanz zurückgeben, um auszudrücken, dass das Ergebnis optional ist.

4.3 Beispielhafte Verwendung [39]

Klasse Person

```
class Person {
    private Person partner;
    Person(Person partner) { this.partner = partner; }
    Person() { this(null); };
    Optional<Person> getPartner() {
        return Optional.ofNullable(partner);
    }
}
```

Bei dem Rückgabetyp `Optional<Person>` von `getPartner()` wird ein Typ-Parameter verwendet, weil `Optional` ein generischer Datentyp ist. Diese werden wir in einem späteren Kapitel betrachten. Hier muss die Information genügen, dass man durch das `<Person>` gestlegen kann, welchen Typ das Objekt hat, welches von `Optional` verwaltet wird.

Verwender

```
Person peter = new Person();
Person klaus = new Person(peter);
Person hans = new Person();
```

```
System.out.println(klaus.getPartner().isPresent()); // -> true
System.out.println(hans.getPartner().isPresent()); // -> false
System.out.println(klaus.getPartner().get()); // -> Person@7530d0a
```

4.4 Primitive Datentypen [40]

Die primitiven Datentypen nehmen in Java immer eine Sonderstellung ein. Wie bereits erläutert, kann man sie mithilfe der *Wrapper-Typen* in Objekte verwandeln. Dies ist aber – wegen des ständigen Ein- und Auspackens – nicht effizient. Deswegen gibt es eine Reihe von speziellen Klassen, um `Optional` einfach für die wichtigsten primitive Datentypen verwenden zu können.

Optional für primitive Datentypen

- `OptionalInt`: Daten vom Typ `int`, `short`, `byte`, `char`
  - `of(int i)`
  - `getAsInt()`
- `OptonalLong`: Daten vom Typ `long`
  - `of(long l)`
  - `getAsLong()`
- `OptonalDouble`: Daten vom Typ `float` und `double`
  - `of(double d )`
  - `getAsDouble()`

Für die anderen Datentypen wird kein Optional angeboten, diese lassen sich aber alle problemlos in den drei vorhandenen speichern.

Im Folgenden ein Beispiel für eine Klasse `Person` bei der das Alter optional angegeben werden kann.

Klasse mit OptionalInt

```
class Student {
    private String name;
    private OptionalInt age;

    public Student(String name, int age) {
        this.name = name;
        this.age = OptionalInt.of(age);
    }
    public Student(String name) {
        this.name = name;
        this.age = OptionalInt.empty();
    }
    public OptionalInt getAge() {
```

```
        return age;
    }
}
```

Die Verwendung von `OptionalInt` ermöglicht es uns, klar auszudrücken, dass das Alter optional ist und dass entsprechende Maßnahmen getroffen werden müssen, um mit dieser Möglichkeit umzugehen.

Verwender

```
Person max = new Student("Max"); // Kein Alter angegeben
Person anna = new Student("Anna", 25); // Alter 25 angegeben

if (max.getAge().isPresent()) {
    System.out.println("Age: " + max.getAge().getAsInt());
}
else {
    System.out.println("Age not available.");
}
```

Kapitel 5

Quiz

5.1 Quiz: Equals I [44]

```
public class A {
    public int i;

    public boolean equals(A obj) {
        return ((obj != null) && (getClass() == obj.getClass())
                && (i == ((A) obj).i));
    }
    public int hashCode() { return i; }

    public static void main(String[] args) {
        A a1 = new A(); a1.i = 5;
        A a2 = new A(); a2.i = 5;
        Object o = a2;

        System.out.println("a1.equals(a2): " + a1.equals(a2));
        System.out.println("a1.equals(o): " + a1.equals(o));
    }
}
```

```
a1.equals(a2): true
a1.equals(o):  false
```

Die equals-Methode von `A` ist fehlerhaft, da sie nicht die korrekte Signatur `boolean equals(Object)` hat. Hierdurch wird die equals-Methode von `Object` nicht (wie gewünscht) überschrieben, sondern überladen. D. h. die Klasse `A` hat jetzt zwei equals-Methoden, nämlich die von `Object` geerbte und die eigene. Die equals-Methode von `A` ist damit überladen worden.

Wie bekannt, erfolgt der Aufruf von überladenen Methoden anhand des statischen Typs, sodass die in `A` implementierte `equals`-Methode für `a1` und `a2` benutzt wird, für `a1` und `o` aber die aus `Object`. Da die Methode aus `Object` keinerlei Rücksicht auf den Inhalt der Objekte nimmt, sondern einen einfachen ==-Vergleich durchführt, kommt es hier zu dem seltsamen Ergebnis.

5.2 Quiz: Equals II [46]

```
public class A {
    public int i;

    public boolean equals(Object obj) {
        return ((obj != null) && (getClass() == obj.getClass())
                && (i == ((A) obj).i));
    }
    public int hashCode() { return i; }
    public static void main(String[] args) {
        A a1 = new A(); a1.i = 5;
        A a2 = new A(); a2.i = 5;

        Object o = a2;

        System.out.println("a1.equals(a2): " + a1.equals(a2));
        System.out.println("a1.equals(o): " + a1.equals(o));
    }
}
```



```
a1.equals(a2): true
a1.equals(o):  true
```



Die equals-Methode von `A` ist jetzt korrigiert worden und überschreibt die entsprechende Methode von `Object`. Dank virtuellem Methodendaufruf funktioniert jetzt alles wie erwartet.

5.3 Quiz: Vergleich [48]

```
Long l1 = 150L;
Long l2 = 200L;

l2 = l2 - 50;

System.out.println(l1 == l2);
```



```
false
```



Wenn man den Source-Code genauer betrachtet fällt auf, dass die Variablen `l1` und `l2` nicht vom primitiven Typ `long` sind, sondern als Referenzvariablen auf den Wrapper-Typ `Long` deklariert wurde. Es handelt sich also um Variablen, die auf echte Objekte zeigen. Die Tatsache,

dass der Source-Code kein `new` Statement enthält, entsteht dadurch, dass der Compiler im Rahmen des Autoboxing automatisch die Objekterzeugung für Wrapper-Typen vornimmt, wenn sie zugewiesen werden.

Wenn es sich aber um echte Objekte handelt, ist ein Vergleich mit == grundsätzlich falsch, wenn man die Objekte auf gleichen Inhalt hin prüfen möchte. == überprüft, ob die Referenzen auf dasselbe Objekt zeigen. Daher gibt das Programmfragment – trotz gleichen Inhalts von `l1` und `l2` `false` aus, da es sich um verschiedene Objekte gleichen Inhalts handelt.

Index

Autoboxing, 12

Class-Objekt, 4

flache Kopie, 7

globale String-Literal-Optimierung, 6

Hash-Kollision, 3

tagging Interface, 8
Tiefe Kopien, 8

Wrapper-Klassen, 11

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Enumerationen und Records



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**
- 1.1 Fragen [5] . . . 1
- 1.2 Begriffe [6] . . . 1
- 1.3 Links zur Java-Insel [7] . . . 1

2 Motivation **2**
- 2.1 Das Problem [9] . . . 2
- 2.2 Beispiel: Das Problem [10] . . . 2
- 2.3 Die Lösung: Enumerationen (Enums) [11] . . . 3
- 2.4 Beispiel: Mit Enumerationen [12] . . . 4

3 Enumerationen **5**
- 3.1 Eigenschaften von Enumerationen [14] . . . 5
- 3.2 Beispiel: Enum mit Konstruktor [15] . . . 6
- 3.3 Beispiel: Iterieren über eine Enum [16] . . . 6
- 3.4 Beispiel: Enums und if / switch [18] . . . 7
- 3.5 Wichtige Methoden von Enumerationen [19] . . . 8
- 3.6 Beispiel: Methoden auf Enums [21] . . . 8
- 3.7 Abstrakte Methoden [23] . . . 9
- 3.8 Beispiel: Abstrakte Methoden [24] . . . 9

4 Records **11**
- 4.1 Motivation [28] . . . 11
- 4.2 Records [30] . . . 12
- 4.3 Konstruktor [32] . . . 13
- 4.4 Methoden [34] . . . 14

5 Quiz **15**
- 5.1 Quiz: Wochentage [36] . . . 15
- 5.2 Quiz: Vergleiche [38] . . . 16

Index **i**

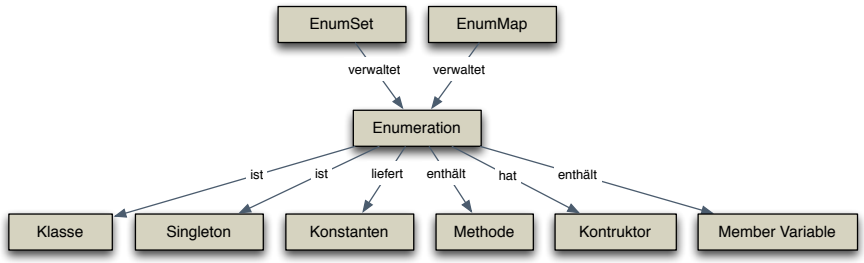
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie kann ich bei Methoden die Auswahl von Werten für die Parameter beschränken?
- Wie kann ich „mal eben schnell“ eine Klasse herbeizaubern?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 6.4 Konstanten und Aufzählungen

Kapitel 2

Motivation

2.1 Das Problem [9]

Wie kann ich …

- … einen festen Satz von Werten vorgeben?
- … dafür sorgen, dass nur diese Werte und kein anderer Wert übergeben werden kann?
- … die Werte typsicher machen?
- … mir dabei die Überprüfung vom Compiler abnehmen lassen?
- … die Werte in gängigen Kontrollstrukturen verwenden?

In der fortgeschrittenen Programmierung benötigt man sehr häufig Methoden, bei denen über einen Parameter eine Auswahl getroffen werden kann, die sich aus einer begrenzten Menge von Möglichkeiten bestimmt. Beispiele sind die Auswahl einer Textauszeichnung (fett, kursiv, unterstrichen) in einer Textverarbeitung oder ein Wochentag in einem Kalender. In beiden Fällen kann man zwar einen `int` verwenden und einfach festlegen, welche Zahl welche Bedeutung haben soll, dies ist aber nicht elegant und vor allem fehleranfällig.

Eigentlich möchte man eine Möglichkeit, die benötigten Konstanten für die Übergabe zu definieren und der Compiler soll sicherstellen, dass beim Aufruf nur eine dieser Konstanten verwendet werden kann (**typsicher**).

typsicher

Schön wäre es zusätzlich noch, wenn man diese Konstanten in allen Kontrollstrukturen (`if` und `case`) verwenden könnte.

2.2 Beispiel: Das Problem [10]

```
public class Wochentag {
    public static final int MONTAG = 1;
    public static final int DIENSTAG = 2;
    public static final int MITTWOCH = 3;
    public static final int DONNERSTAG = 4;
    public static final int FREITAG = 5;
    public static final int SAMSTAG = 6;
    public static final int SONNTAG = 7;
```

```
}
public class DateJuggler {
    public void setWochentag(int wochentag) { ... }
    public int getWochentag() { ... }
}

DateJuggler dj = new DateJuggler();
dj.setWochentag(Wochentag.FREITAG);
dj.setWochentag(Wochentag.FREITAG + 100);
int wochentag = dj.getWochentag();
```

Das Beispiel zeigt das Problem, wenn man keine Unterstützung für für die oben erwähnten Konstanten in der Programmiersprache hat. Es gibt eine ganze Reihe von Fehlerquellen:

1. Bei der Definition der Konstanten (`MONTAG`, `DIENSTAG`, …) muss man aufpassen, dass man keinen der Werte doppelt verwendet. Würde man `SONNTAG` aus Versehen ebenfalls auf `6` setzen, käme es zu subtilen, schwer zu findenden Fehlern.
2. Man kann nicht sicherstellen, dass der Verwender nur eine der vorhandenen Konstanten verwendet. Da die Methode `setWochentag` einen `int` nimmt, kann man jede Zahl übergeben, auch unsinnige Tage wie `10` oder `FREITAG + 100`. 3. Innerhalb der Methode `setWochentag` muss man also durch eine Überprüfung noch einmal feststellen, ob der übergebende Wert im gewünschten Bereich liegt. Wenn nicht, muss man entweder eine Ausnahme werfen oder aber einen Standardwert annehmen.

2.3 Die Lösung: Enumerationen (Enums) [11]

Enumeration (enumeration) Enumeration

- Konstrukt, um festen Satz von Konstanten zu definieren
- Typsicher und vom Compiler überprüft
- Funktionieren mit `if` und `switch`
- Syntax: `[Sichtbarkeit] enum { CONST1, CONST2, ... ; }`

```
public enum Planet {
    MERKUR, VENUS, ERDE, MARS,
    JUPITER,SATURN, URANUS, NEPTUN;
    // PLUTO;
}
```

Java bietet mit den sogenannten Enumerationen eine Lösung für das beschriebene Problem. Mit ihnen kann man einen festen Satz von Konstanten definieren, wobei der Compiler überprüft, dass ein Verwender nur die erlaubten Konstanten übergeben kann.

In dem hier gegebenen Beispiel sehen wir eine Enumeration (**Enum**) mit dem Namen `Planet`, die acht Enum-Konstanten (nämlich die Planeten) enthält. Enum

Technisch gesehen ist eine Enumeration einer Klasse sehr ähnlich. Sie führt einen neuen Typ ein (hier `Planet`) und erzeugt automatisch Objekte für die vorhandenen Konstanten (hier `MERKUR`, `VENUS`, `ERDE`, ...). Sie wird hinter den Kulissen tatsächlich als Klasse realisiert, wobei es aber einige Besonderheiten gibt, die später noch erläutert werden. Zu jeder Enumeration gibt es also:

1. Eine Klasse für die Enumeration
2. Genau ein Objekt pro Konstante, die in der Enumeration deklariert wurde

Da eine Enumeration einen neuen Typ einführt, kann man von ihrem Typ Referenzvariablen deklarieren und damit auf die Konstanten-Objekte zeigen: `Planet p = Planet.MERKUR`

2.4 Beispiel: Mit Enumerationen [12]

```
public enum Wochentag {
    MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG, SAMSTAG, SONNTAG;
}

public class DateJuggler {
    public void setWochentag(Wochentag wochentag) { ... }
    public Wochentag getWochentag() { ... }
}

DateJuggler dj = new DateJuggler();
dj.setWochentag(Wochentag.FREITAG);
dj.setWochentag(Wochentag.FREITAG + 100); // FEHLER!!

Wochentag wochentag = dj.getWochentag();
```



Dieses Beispiel zeigt noch einmal die Klasse `DateJuggler`, jetzt aber mithilfe von Enumerationen realisiert. Die Konstanten befinden sich jetzt in der neu geschaffenen Enumeration `Wochentag`. Die `setWochentag`-Methode bekommt keinen `int` mehr übergeben, sondern eine der Konstanten aus `Wochentag`, was durch den Typ des Parameters angezeigt wird.

Der Verwender von `setWochentag` kann jetzt die Methode nur noch mit einer der vorhandenen Konstanten aufrufen, jeder Versuch etwas anderes zu übergeben führt zu einem Fehler.

Kapitel 3

Enumerationen

3.1 Eigenschaften von Enumerationen [14]

- Können *nicht* von anderen Klasse ableiten, sondern werden vom Compiler automatisch von `Enum` abgeleitet
- Können Felder, Konstanten und Methoden enthalten
- Enum-Konstanten sind echte *Java-Objekte*
- Enum-Konstanten sind *Singletons*
  - es gibt zu jeder Enum-Konstante immer nur ein einziges Objekt
  - dürfen mit == verglichen werden
  - können nicht mit `new` erzeugt werden
  - können nur `private` oder default-sichtbare Konstruktoren haben
- Enum-Konstanten können statisch importiert werden

`Enum`

Alle Enumerationen sind automatisch von `java.lang.Enum` abgeleitet. Daher kann man in der Deklaration einer Enumeration auch kein `extends` angeben. Die Klasse heißt *Enum* und nicht `Enumeration`, weil es bei ihrer Einführung bereits eine Klasse mit dem Namen `Enumeration` gab und man diesen daher nicht mehr verwenden konnte.

Der **Konstruktor einer Enumeration** muss private oder default-Sichtbarkiet haben. Dies ist nötig, da andernfalls die Eigenschaft der Konstanten als **Singleton** unterwandert würde.

Konstruktor einer Enumeration

Singleton

Wie man an dieser Aufzählung bereits sieht, gehen die Eigenschaften von Java-Enumerationen weit über das hinaus, was in anderen Programmiersprachen möglich ist. So werden z. B. in C# und C++ Enumerationen einfach auf int-Werte abgebildet, wohingegen in Java das sog. „type safe Enumeration Pattern“ mit echten Objekten und einem reichhaltigen Satz von Features zur Verfügung steht.

Einige der Möglichkeiten, die Enumerationen bieten sind obskur, so die Möglichkeit abstrakte Methoden zu definieren (wird später noch erläutert). Man muss – und sollte – diese aber nicht unbedingt verwenden. Davon abgesehen sind Enumerationen sehr nützlich.

3.2 Beispiel: Enum mit Konstruktor [15]

```
public enum Planet {
    MERKUR(59e+6, 88), VENUS(108e+6, 225),
    ERDE(150e+6, 365), MARS(228e+6, 1.9*365),
    JUPITER(778e+6, 11.9*365), SATURN(1427e+6, 29.5*365),
    URANUS(2870e+6, 84*365), NEPTUN(4497e+6, 165*365);

    private final double entfernungsonne; // in km
    private final double umlaufzeit;  // in tagen

    private Planet(double entfernungsonne, double umlaufzeit) {
        this.entfernungsonne = entfernungsonne;
        this.umlaufzeit = umlaufzeit;
    }

    public double bahngeschwindigkeit() {
        return 2 * entfernungsonne * Math.PI / umlaufzeit / 24;
    }
}
```

Man kann einer Enumeration Daten mitgeben, d. h. die Konstanten tragen dann neben ihrem Namen noch beliebige weitere Informationen. Dies kann sehr nützlich sein, wenn es Werte gibt, die direkt zu den Konstanten gehören. Im Beispiel sind das die Bahngeschwindigkeit der Planeten und ihre Entfernung zur Sonne.

Die Enum-Konstanten (z. B. `MERKUR`) geben dann bei ihrer Deklaration die Werte an, die in den Attributen des Objektes gespeichert werden sollen. Technisch gesehen handelt es sich also um einen Konstruktoraufruf, nur dass das `new`

implizit erzeugt wird.

3.3 Beispiel: Iterieren über eine Enum [16]

```
public class Ausgabe {

    public static void main(String[] args) {

        for (Planet planet : Planet.values()) {
            System.out.printf("%s -> %,.0f km/h%n",
                    planet.name(),
                    planet.bahngeschwindigkeit());
        }
    }
}
```

```
MERKUR -> 175.525 km/h                    VENUS -> 125.664 km/h
ERDE -> 107.589 km/h                      MARS -> 86.071 km/h
JUPITER -> 46.893 km/h                    SATURN -> 34.696 km/h
URANUS -> 24.506 km/h                     NEPTUN -> 19.549 km/h
```

Jede Enumeration hat eine statische Methode `values()`, die ein Array mit allen vorhandenen Konstanten zurückliefert. Will man also ein Programm schreiben, dass über alle Konstanten iteriert und dann nicht verändert werden muss, wenn man eine weitere Konstante einführt, bietet sich diese Methode an. Über `name()` kann man dann den Namen der Konstante wieder auslesen.

`values()`

`name()`

Das Beispiel nutzt genau dies aus. Es läuft mit einer for-each-Schleife über alle Konstanten, liest dann den Namen aus und ruft eine Methode (`bahngeschwindigkeit()`) auf.

3.4 Beispiel: Enums und if / switch [18]

```
public class SpaceTravel {
    private static final Planet HOME_PLANET = Planet.ERDE;

    public void travel(Planet zielPlanet) {

        if (zielPlanet == HOME_PLANET) {
            return; // wir sind doch schon da
        }

        switch (zielPlanet) {
            case JUPITER: // langer Flug
                break;
            case MARS:   // kurzer Flug
                break;
             ...
        }
    }
}
```

Da es sich bei Enumerationen um Singletons handelt, darf man sie – anders als alle anderen Objekte in Java – mit == vergleichen. Zusätzlich besteht noch die Möglichkeit, sie in switch-Verzweigungen zu verwenden.

Beachtenswert ist, dass man bei den case-Statements nur den Namen der Konstante ohne vorangestellten Namen der Enumeration benutzen muss. Der Grund liegt darin, dass bereits aufgrund des Typs der Variable im `switch` klar ist, um welche Enumeration es sich handelt (`zielPlanet` ist vom Typ `Plant`, damit ist klar, wo der Compiler nach `JUPITER` und `MARS` suchen muss).

3.5 Wichtige Methoden von Enumerationen [19]

- `public final String name()` – Liefert den Namen der Konstante, genauso wie sie in der Datei angegeben wurde
- `public final int ordinal()` – Liefert die Ordinalzahl der Konstante (sollte normalerweise nicht verwendet werden)
- `public String toString()` – Liefert den Namen der Konstante, kann überschrieben werden, um besser lesbare Namen zurückzugeben
- `public static Enum valueOf(String name)` – Wandelt den Namen der Konstanten (wie von `name()` zurückgeliefert) wieder in eine Enum um
- `public static Enum[] values()` – Gibt ein Array mit allen vorhandenen Enum-Konstanten zurück
- `public final int compareTo(Enum o)` – Vergleicht Enums anhand ihrer Ordinalzahlen
- `public final boolean equals(Object other)` – Entspricht `Object.equals()`und damit ==
- `public final int hashCode()` – Entspricht `Object.hashCode()`
- `protected final Object clone()` – Darf nicht verwendet werden, da Enums Singletons sind

Da alle Enumerationen von der Klasse `Enum` abgeleitet sind, erben sie entsprechende Methoden. Die meisten Methoden sind sofort verständlich. Wichtig ist der Unterschied zwischen `name()` und `toString()`. Beide liefern standardmäßig den Namen der Konstante, wie er in der Enumeration deklariert wurde, zurück. `toString()` kann man aber überschreiben, `name()` nicht. Insofern kann man `toString()` verwenden, um einen eigenen Namen zurückzugeben während `name()` immer unverändert bleibt und exakt den Namen der Konstante entsprechend ihrer Deklaration zurück gibt.

Da es sich bei Enumerationen immer um Singletons handelt, sollten die Methoden `hashCode()`, `equals()`, `clone()` und `compareTo()` nicht verändert werden. Deshalb sind sie als `final` deklariert, um ein versehentliches Überschreiben zu verhindern.

3.6 Beispiel: Methoden auf Enums [21]

```
String name = Planet.ERDE.name();

Planet planet = Planet.valueOf(name);

System.out.printf("Wir leben auf der %s, sie ist " +
       "der %d Planet%n", planet, planet.ordinal() + 1);

System.out.printf("Wir haben %d Planeten im Sonnensystem%n",
       Planet.values().length);
```



```
Wir leben auf der ERDE, sie ist der 3 Planet
Wir haben 8 Planeten im Sonnensystem
```

Das Beispiel verwendet die eingebauten Methoden, die jede Enumeration hat. Es wird gezeigt, dass man den Namen einer Konstante (`Planet.ERDE.name()`) über die statische `valueOf()`-Methode wieder in eine Referenz auf die Konstante umwandeln kann.

`valueOf()`

Man sieht auch, dass die von `ordinal()` gelieferten Werte bei 0 beginnen, sodass im Beispiel noch eine 1 addiert werden muss.

`ordinal()`

3.7 Abstrakte Methoden [23]

- Enums können **abstrakte Methoden** enthalten
- Diese Methoden müssen für jede Enum-Konstante implementiert werden
- Mit dieser Konstruktion sollen unübersichtliche case bzw. if-Konstrukte vermieden werden

abstrakte Methoden

Die Möglichkeit in einer Enumeration abstrakte Methoden zu definieren, um diese dann in den Konstanten zu implementieren, ist ausgesprochen obskur und verwirrend. Wenn man sich Enumerationen als Klassen und die Konstanten als Objekte vorstellt bedeutet dies, dass die Objekte (sic!) die Methoden implementieren, die die Klasse als abstrakt definiert hat. Aus objektorientierter Sicht ist dies unsinnig, da nur Klassen, aber nicht Objekte Methoden tragen können.

Technisch löst Java diesen Widerspruch indem für jede Konstante eine eigene Subklasse der Enumeration generiert wird und diese dann die entsprechende Methodenimplementierung enthält.

Wegen seiner seltsamen Syntax sollte man dieses Feature nur einsetzen, wenn es einen wirklichen Vorteil bringt.

3.8 Beispiel: Abstrakte Methoden [24]

```
public enum Operation {
    ADDITION {
        public double anwenden(double op1, double op2) {
            return op1 + op2;
        }
    },
    SUBTRAKTION {
        public double anwenden(double op1, double op2) {
            return op1 - op2;
        }
    },
    MULTIPLIKATION {
```

```
        public double anwenden(double op1, double op2) {
            return op1 * op2;
        }
    };
    public abstract double anwenden(double op1, double op2);
}
```

Das Beispiel zeigt, wie die Konstante die abstrakte Methode anwenden implementieren. Abhängig von der Konstante führt die Methoden unterschiedliche Rechenoperationen durch.

```
public class Rechner {

    public double rechne(double op1, double op2, Operation operation) {
        return operation.anwenden(op1, op2);
    }

    public static void main(String[] args) {
        Rechner r = new Rechner();
        double summe = r.rechne(5, 5, Operation.ADDITION);
        double differenz = r.rechne(6, 3, Operation.SUBTRAKTION);
        double produkt = r.rechne(6, 3, Operation.MULTIPLIKATION);
        System.out.printf("%.0f, %.0f, %.0f", summe, differenz, produkt);
    }
}
```

```
10, 3, 18
```

Kapitel 4

Records

4.1 Motivation [28]

Menschen, die andere Programmiersprachen kennen sind häufig davon erstaunt, wie viel Aufwand man in Java betreiben muss, um in einer Klasse ein paar Daten zu speichern. Will man beispielsweise ein Klasse für Rechtecke schreiben, so kommt der folgende Code dabei heraus:

```
public final class Rectangle {

    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double length() { return this.length; }
    public double width() { return this.width; }

    @Override
    public boolean equals(Object o) {
        // ...
    }
```



```
    @Override
    public int hashCode() {
        // ...
    }

    @Override
    public String toString() {
        return "Rechteck{" + "length=" + length + ", width=" + width + '}';
    }
```



```
}
```

Nutzung

```
Rectangle r = new Rectangle(4, 3);
System.out.println("Fläche: " + r.length() * r.width());
```

Man sieht sofort, dass der Schreibaufwand erheblich ist, obwohl es sich hier um einen absolut primitiven Datencontainer handelt.

Um das Schreiben von einfachen Datenklassen zu vereinfachen, wurde mit Java 14 die *Records* eingeführt.

4.2 Records [30]

Records

Records

- wurden mit Java 14 eingeführt
- bieten eine einfache Möglichkeit, Datenklassen zu definieren
- Syntax: `record <NAME> (<ATTRIBUTE>){ <RUMPF> }`

```
record Rectangle(double length, double width) { }
```

Nutzung

```
Rectangle r = new Rectangle(4, 3);
System.out.println("Fläche: " + r.length() * r.width());
```

Das Ergebnis dieser Deklaration ist identisch mit dem umfangreichen Java-Code, der im Beispiel davor gegeben wurde. Im Hintergrund werden vom Java-Compiler die notwendigen Methoden ergänzt.

Generiert werden

- Klasse (`final`), abgeleitet von `java.lang.Record`
- Attribute (`final` und `private`)
- Konstruktor mit passender Signatur
- Zugriffsmethoden, die ebenso heißen, wie die Attribute
- `equals`- und `hashCode`-Methode
- `toString`-Methode

Records sind `final`, man kann also nicht von ihnen erben

4.3 Konstruktor [32]

Der Konstruktor von Records wird automatisch erstellt und als **canonical constructor** bezeichnet. Er nimmt Parameter, die den Attributen bei der Definition des Records entsprechen und weist dann die Werte zu.

canonical constructor

Wenn man die Attribute prüfen möchte, bevor man sie zuweist, kann man einen eigenen Konstruktor schreiben, der eine entsprechende Prüfung durchführt.

- Eigener *canonical constructor* mit Prüfung der Parameter

```
record Rectangle(double length, double width) {
    public Rectangle(double length, double width) {
        if (length <= 0 || width <= 0) {
            throw new java.lang.IllegalArgumentException(
                    String.format("Invalid dimensions: %f, %f", length, width));
        }
        this.length = length;
        this.width = width;
    }
}
```

Wenn man sich das Beispiel ansieht, fällt auf, dass die Attribute in der Record-Deklaration und im Konstruktor zwei mal angegeben werden und damit wieder unnötige Schreibarbeit entsteht. Aus diesem Grund, erlauben es Records, diese Dopplung einfach wegzulassen.

compact constructor

compact constructor

- Wiederholung der Parameterliste entfällt
- Zuweisung der Attribute wird weggelassen

```
record Rectangle(double length, double width) {
    public Rectangle {
        if (length <= 0 || width <= 0) {
            throw new java.lang.IllegalArgumentException(
                    String.format("Invalid dimensions: %f, %f", length, width));
        }
    }
}
```

Bei dieser Form entfällt die Zuweisung der Konstruktor-Parameter an die Attribute und wird vom Compiler automatisch ergänzt.

4.4 Methoden [34]

Da Records im Endeffekt normale Java-Klassen sind, bei denen der Compiler uns einen großen Teil der Arbeit abnimmt, kann man sie auch mit eigenen Methoden versehen.

Man kann eigene Methoden schreiben

- um die automatisch generierten zu ersetzen
- um neue Methoden hinzuzufügen (auch `static`)

```
record Rectangle(double length, double width) {
    public double length() {
        return length;
    }
}
```

Das obige Beispiel ist zwar nicht besonders sinnvoll, zeigt aber die Möglichkeit, die bereits existierenden Zugriffsmethode `length()` durch eine eigene Implementierung zu ersetzen.

```
record Rectangle(double length, double width) {
    public double area() {
        return length * width;
    }
}
```

Kapitel 5

Quiz

5.1 Quiz: Wochentage [36]

```
public enum Wochentage {
    MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG,
    FREITAG, SAMSTAG, SONNTAG(false);

    private final boolean werktag;

    public Wochentage() {
        werktag = true;
    }

    public Wochentage(boolean werktag) {
        this.werktag = werktag;
    }
}
```



```
public enum Wochentage {
    MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG,
    FREITAG, SAMSTAG, SONNTAG(false);

    private final boolean werktag;

    public Wochentage() { // FEHLER!!
        werktag = true;
    }

    public Wochentage(boolean werktag) { // FEHLER!!
        this.werktag = werktag;
    }
}
```



Das Beispiel kompiliert erst gar nicht, weil Enumerationen keine öffentlichen Konstruktoren haben dürfen. Da es sich bei den Konstanten um Singletons handelt, darf es nicht möglich sein von außen weitere Objekte zu erzeugen.

5.2 Quiz: Vergleiche [38]

```
public enum Geschlecht {
    MAENNLICH, WEIBLICH;
}
```



```
Geschlecht g1 = Geschlecht.valueOf("MAENNLICH");
Geschlecht g2 = Geschlecht.values()[0];
Geschlecht g3 = Geschlecht.MAENNLICH;

System.out.println(g1 == g2);
System.out.println(g1 == g3);
System.out.println(g1.equals(g3));
```



```
true
true
true
```



Für Enumerationen garantiert die VM, dass immer nur ein einziges Objekt pro Enum-Konstante (hier also `MAENNLICH` und `WEIBLICH`) vorhanden ist. Daher darf man Enum-Objekte auch, im Gegensatz zu allen anderen Objekten, mit == vergleichen. Hierbei ist es egal, wie man an das jeweilige Enum-Objekt gelangt ist.

Index

abstrakte Methoden, 9

canonical constructor, 13
compact constructor, 13

Enum, 3
Enumeration, 3

Konstruktor einer Enumeration, 5

Records, 12

Singleton, 5

typsicher, 2

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Ausnahmebehandlung



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**
- 1.1 Fragen [5] . . . 1
- 1.2 Begriffe [6] . . . 1
- 1.3 Links zur Java-Insel [7] . . . 1

2 Motivation: Warum Ausnahmen? **2**
- 2.1 Motivation: Gängige Fehlersituationen [9] . . . 2
- 2.2 Das Problem [10] . . . 2
- 2.3 Beispiel: Fehlerbehandlung in C [11] . . . 3
- 2.4 Anforderungen an Fehlerbehandlung [12] . . . 4

3 Try-Catch **5**
- 3.1 Beispiel: Ausnahme [14] . . . 5
- 3.2 Fehlerbehandlung in Java [16] . . . 6
- 3.3 Beispiel: try und catch [17] . . . 6
- 3.4 Mehrere catch-Blöcke [18] . . . 7
- 3.5 Beispiel: Mehrere catch-Blöcke [19] . . . 8
- 3.6 Multi-Catch [21] . . . 8
- 3.7 So nicht! [22] . . . 9
- 3.8 Vergleich zu C [23] . . . 10

4 Finally **12**
- 4.1 finally-Block [26] . . . 12
- 4.2 Beispiel: finally-Block [27] . . . 13
- 4.3 Ausnahmen im finally-Block [28] . . . 13
- 4.4 Abruptes Beenden des finally-Blocks [29] . . . 14
- 4.5 try-with-resources Statement [31] . . . 15

5 Call-Stack **17**
- 5.1 Der Call-Stack-Mechanismus [33] . . . 17
- 5.2 Beispiel: Call-Stack-Mechanismus [34] . . . 17

6 Arten von Ausnahmen **21**
- 6.1 Kategorien von Ausnahmen [39] . . . 21
- 6.2 Hierarchie der Java-Ausnahmen [40] . . . 22
- 6.3 Die Klasse Throwable [41] . . . 22

6.4 Die Klasse Exception [42] . . . 23
6.5 Beispiele für häufige Ausnahmen [43] . . . 24
6.6 Die handle or declare Regel [44] . . . 24
6.7 Beispiel: Handle or Declare [45] . . . 25
6.8 Ausnahmen und Überschreiben [47] . . . 26
6.9 Beispiel: Überschreiben und Exceptions [48] . . . 26
6.10 Ausnahmen und Programmierstil [49] . . . 27
6.11 Beispiel: Schlechter Stil [50] . . . 27

7 Eigene Ausnahmen 29
7.1 Ausnahmen selbst schreiben [52] . . . 29
7.2 Ausnahmen selbst werfen [53] . . . 29
7.3 Beispiel: Exceptions werfen [54] . . . 30
7.4 Beispiel: Die Exception [58] . . . 31
7.5 Beispiel: Der Werfer [59] . . . 32
7.6 Beispiel: Der Fänger [60] . . . 32

8 Ausnahmen und JUnit 34
8.1 Wie teste ich auf Ausnahmen? [62] . . . 34
8.2 Beispiel: try und catch [63] . . . 34
8.3 Beispiel: assertThrows [64] . . . 35

9 Quiz 36
9.1 Quiz: Ausnahmen und Überschreiben [66] . . . 36
9.2 Quiz: finally [68] . . . 37

Index ii

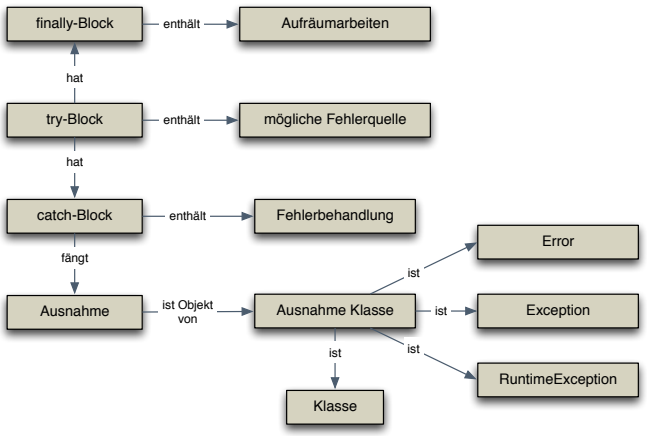
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie kann ich Fehler sinnvoll behandeln?
- Wie kann ich Fehler anzeigen und verarbeiten?
- Wie kann ich Programme mit potenziellen Fehlern testen?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 9. Ausnahmen müssen sein

Kapitel 2

Motivation: Warum Ausnahmen?

2.1 Motivation: Gängige Fehlersituationen [9]

- Datei soll geladen werden, existiert aber nicht
- Netzwerkverbindung ist nicht verfügbar
- String „a34b“ soll in einen Integer umgewandelt werden
- Programm versucht 15 / 0 zu berechnen
- Methode greift hinter das letzte Element eines Arrays
- Es ist kein Speicher mehr auf dem Heap vorhanden
- Ein Programm gerät in eine unendliche Rekursion und verbraucht den gesamten Stack-Speicher
- Methode versucht, einen String auf einen Integer zu casten

Dass in Programmen Fehler auftreten, ist ganz normal. Häufig liegt die Ursache für ein Problem nicht im Programm selbst, sondern in der Umgebung oder den Eingaben des Benutzers. So gibt der/die Benutzer:in möglicherweise einen falschen Dateinamen an oder das Netzwerk funktioniert während eines entfernten Zugriffs nicht richtig.

Insofern muss sich eine gute Software immer um Fehlersituationen kümmern. Es ist sogar so, dass häufig noch ein erheblicher Aufwand zwischen einer Software, die die gewünschte Funktionalität ausführt und einer Software, die in Fehlersituationen richtig funktioniert, liegt.

2.2 Das Problem [10]

Wie transportiert man die Information, dass ein Fehler aufgetreten ist dorthin wo er korrigiert werden kann?

- Sofortiger Abbruch des Programms?
- Setzen einer globale Fehlervariable?
- Magischer Rückgabewert?
- Fehler ignorieren und protokollieren?
- Aufruf eines Callbacks zur Fehlerbehandlung?
- Einfach Standardwerte annehmen und weitermachen?

Keine der hier aufgezeigten Methoden ist besonders befriedigend. Trotzdem kommen sie bei vielen Programmiersprachen zum Einsatz. So ist das sofortige Abbrechen des Programms der Standardfall in vielen Sprachen (auch in Java), wenn sich der/die Programmierer:in nicht um den Fehler kümmert.

Das Setzen einer Fehlervariable und magische Rückgabewerte kommen in C zum Einsatz. Visual Basic bis Version 6 bietet mit `On Error Goto` sowohl Callbacks als auch mit `On Error Resume Next` die Möglichkeit, Fehler einfach zu ignorieren und beim nächsten Statement weiterzuarbeiten. Es ist offensichtlich, dass das Ignorieren von Fehlern keine ernsthafte Fehlerbehandlung darstellt und Probleme oft noch verschlimmert.

2.3 Beispiel: Fehlerbehandlung in C [11]

```
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
    DieWithError("socket () failed") ;
}
/* Bind to the local address */
if (bind(servSock, (struct sockaddr *)&echoServAddr,
       sizeof(echoServAddr)) < 0) {
   DieWithError ("bind () failed");
}
/* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0) {
   DieWithError("listen() failed") ;
}
for (;;) { /* Run forever */
   if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr,
          &clntLen)) < 0) { /* Wait for a client to connect */
      DieWithError("accept() failed");
   }
}
```

Man sieht hier an dem C-Beispiel sehr deutlich die Probleme klassischer Fehlerbehandlung über Rückgabewerte:

- Die normale Programmlogik und die Fehlerbehandlung sind wild gemischt.
- Der Programmcode wird erheblich aufgebläht und sehr unübersichtlich, da manche IFs der Fehlerbehandlung dienen und andere wieder dem normalen Kontrollfluss.
- Der Rückgabewert von Funktionen ist doppeldeutig: er transportiert sowohl Fehler- als auch normale Informationen.
- Es kann leicht passieren, dass man vergisst einen Fehlercode abzufragen, sodass Fehler überhaupt nicht behandelt werden.
- Es ist sehr schwer Fehler weiterzureichen, da die hier dargestellte Funktion vier verschiedene Fehlersituationen im eigenen Rückgabewert codieren müsste.

2.4 Anforderungen an Fehlerbehandlung [12]

- Fehlerbehandlung und normaler Betrieb sind orthogonal
- Fehler können nicht ignoriert werden und werden garantiert behandelt
- Fehlerbehandlung orientiert sich an der Aufrufhierarchie
- In Aufrufhierarchie kann an beliebiger Stelle reagiert werden
- Fehlerbehandlung soll objektorientiert sein
- Eigene Fehler können definiert und verarbeitet werden

- *Fehlerbehandlung und normaler Betrieb sind orthogonal* – Die Fehlerbehandlung und die normale Programmlogik sollen sich im Programm nicht vermischen, sondern es soll jederzeit klar erkennbar sein, bei welchen Statements es sich um Fehlerbehandlung handelt und bei welchen nicht. Weiterhin sollen normale Logik und Fehlerbehandlung an getrennten Stellen stehen und sich nicht Zeile für Zeile mischen.
- *Fehler können nicht ignoriert werden* – Es soll den Programmierer:innen nicht möglich sein, Fehlersituationen einfach zu ignorieren und Fehler nicht abzufragen. Gerade Fehler, die nicht behandelt werden, führen häufig zu erheblichen Problemen, weil das Programm auf einem Bein weiter humpelt, um dann an einer ganz anderen Stelle (fern ab der eigentlichen Ursache) zusammenzubrechen. Hierdurch sucht man dann den Fehler an einer völlig falschen Stelle.
- *Fehlerbehandlung orientiert sich an der Aufrufhierarchie* – Häufig liegt die Ursache für einen Fehler nicht an der Stelle, an der er auftritt. Gibt z. B. ein/eine Benutzer:in einen falschen Dateinamen an, so tritt der Fehler im Programm an der Stelle auf, an der die Datei geöffnet werden soll, die Ursache ist aber die falsche Eingabe. Daher möchte man Fehler die Aufrufhierarchie hinauf transportieren können, bis sie beim Verursacher ankommen.
- *In Aufrufhierarchie kann an beliebiger Stelle reagiert werden* – Natürlich möchte man beim Transport entlang der Aufrufhierarchie an jeder beliebigen Stelle reagieren können.
- *Fehlerbehandlung soll objektorientiert sein* – In einer objektorientierten Sprache wie Java wäre es armselig, wenn man nicht die Fehlerbehandlung nach objektorientierten Kriterien aufbauen würde. Andernfalls wäre die Fehlerbehandlung ein Fremdkörper in der Sprache.
- *Eigene Fehler können definiert und verarbeitet werden* – Der/die Programmierer:in sollte in der Lage sein, eigene Fehlerarten entwerfen und behandeln zu können.

Kapitel 3

Try-Catch

3.1 Beispiel: Ausnahme [14]

```java
public class ExceptionExample {

    public static void main(String[] args) {

        String[] namen = { "Franz", "Hans", "Alfons" };

        for (int i = 0; i < 4; i++) {
            System.out.print(namen[i] + ", ");
        }
    }
}
```



```
Exception in thread "main" ↩
    java.lang.ArrayIndexOutOfBoundsException: 3
     at pr2.ExceptionExample.main(ExceptionExample.java:10)
Franz, Hans, Alfons,
```



Das Beispiel zeigt, was passiert, wenn in Java eine Ausnahme auftritt. Das Programm versucht hinter das letzte Element des Arrays zu greifen, was von der Laufzeit erkannt wird und zum Auslösen einer `ArrayIndexOutOfBoundsException` führt.

Man sieht an dem Beispiel schon sehr schön die Vorteile der Fehlerbehandlung in Java: Der Typ der Ausnahme weist direkt durch seinen Namen darauf hin, welchen Fehler man gemacht hat. Zusätzlich enthält die Ausnahme die Information, welchen ungültigen Array-Index man verwendet hat (3). Der Call-Stack zeigt die Datei und die Zeilennummer an, in der der Fehler aufgetreten ist.

3.2 Fehlerbehandlung in Java [16]

Ausnahmen

- Fehler werden durch eine **Ausnahmen** (exceptions) signalisiert
  - Exceptions sind Java Objekte, die im Fehlerfall erzeugt werden
  - Alle Exceptions sind (indirekt) von `Throwable` abgeleitet

try-Block

- **try-Block** enthält Code, der mit Fehlerbehandlungsroutine(n) versehen werden soll
  - Tritt ein Fehler auf, wird die Fehlerbehandlungsroutine angesprungen (`try`-Block wird **abrupt beendet**)

abrupt beendet

  - Tritt kein Fehler auf, läuft das Programm nach dem try/catch weiter (`try`-Block wird **normal beendet**)

normal beendet

catch-Blöcke
Fehlerbehandlungsroutin

- **catch-Blöcke** enthalten die **Fehlerbehandlungsroutine** (exception handler) zum `try`-Block
  - Fehler wird als Objekt an `catch` übergeben

Das Exception-Konzept von Java ist stark an die C++-Exceptions angelehnt.

Exception
Throwable

Jeder Fehler (**Exception**) wird durch ein Objekt repräsentiert, dessen Klasse von `Throwable` abgeleitet ist. Der Typ des Objektes liefert eine genaue Information, welche Art von Fehler aufgetreten ist.

try

Die Fehlerbehandlung wird von der normalen Programmlogik dadurch getrennt, dass man zwei unterschiedliche Blöcke verwendet. Der eine Block wird mit `try` eingeleitet und enthält die Anweisungen, die möglicherweise zu einem Fehler führen könnten, z. B. Zugriff auf eine Datei. Der andere Block wird mit `catch` gekennzeichnet und enthält die Fehlerbehandlung. Er wird nur dann angesprungen, wenn im `try`-Block ein Fehler aufgetreten ist. Wenn ein solcher Fehler auftritt, wird der `try`-Block *abrupt* beendet, d. h. er wird nicht weiter ausgeführt, sondern die Programmausführung springt zu dem `catch`-Block. Man sagt auch, dass der `catch`-Block die Ausnahme fängt (**Fangen einer Ausnahme**). Wenn kein Fehler auftritt, läuft der `try`-Block ganz normal zu Ende und der `catch`-Block wird ignoriert. D. h. die Programmausführung geht nach dem `catch`-Block weiter.

catch

Fangen einer Ausnahme

Über einen Parameter wird dem `catch`-Block das Ausnahmeobjekt übergeben, auf das er dann über eine lokale Referenzvariable zugreifen kann. Nachdem der `catch`-Block am Ende angekommen ist, wird die Programmausführung nach dem `catch`-Block fortgesetzt.

3.3 Beispiel: try und catch [17]

```
Connection connection = getConnection();

try {
    ResultSet rs = readData(connection);

    while (rs.next()) {
        // Daten bearbeiten
    }
```

```
} catch (SQLException ex) {
    // Datenbank Problem behandeln
    String sqlStatus = ex.getSQLState();
    ex.printStackTrace();
    ...
}

...
```

Dieses Beispiel zeigt, wie eine potenziell auftretende `SQLException` behandelt werden kann – hierbei handelt es sich um einen Fehler, der bei der Kommunikation mit einer Datenbank auftreten kann. Im `try`-Block befindet sich der Datenbankzugriff, der sich hinter der Methode `readData()` und den Zugriffen auf das `ResultSet` mit `rs.next()` verbirgt. Bei jedem Datenbankzugriff kann potenziell ein Problem entstehen, z. B. ist die Datenbank nicht verfügbar, das dann durch eine `SQLException` signalisiert wird.

Die Behandlung des Datenbankfehlers erfolgt in diesem Beispiel im entsprechenden `catch`-Block, der die `SQLException` fängt. Obwohl die eigentliche Fehlerbehandlung hier weggelassen wurde, kann man eine weitere wichtige Eigenschaft von Exceptions erkennen: Dadurch, dass sie echte Java-Objekte sind, kann man über die Ausnahme zusätzliche Daten transportieren. Hier den Fehler-Code der Datenbank, der mit `getSQLState()` ausgelesen wird und den Stack-Trace, der mit `printStackTrace()` ausgedruckt wird.

Die hier dargestellte Fehlerbehandlung ist nur als Beispiel zu verstehen und natürlich nicht realistisch. Vor allem das Ausdrucken des Stack-Traces ergibt in echten Systemen nur wenig Sinn.

3.4 Mehrere catch-Blöcke [18]

- Man kann *beliebig viele* `catch`-Blöcke angeben
- Die Laufzeit wählt den Block aus, bei dem die angegebene Ausnahme am besten zur tatsächlich geworfenen passt
- Die Auswahl geschieht *anhand der Vererbungshierarchie* der Ausnahmen
- Die Blöcke müssen in der Reihenfolge *vom Spezifischen zum Allgemeinen* angegeben werden

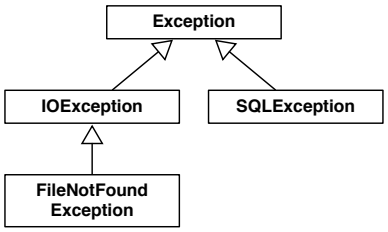
In einem `try`-Block können durchaus verschiedene Ausnahmen auftreten, die man unterschiedlich behandeln möchte. Daher ist es möglich, einem `try`- mehrere `catch`-Blöcke zuzuordnen.

Die Auswahl des `catch`-Blockes erfolgt dann zur Laufzeit anhand des Typs der Ausnahme, die im `try`-Block geworfen wurde. Ähnlich zu überladenen Methoden wird also der am besten passenden Block ausgewählt. (Anders als bei überladenen Methoden aber erst zur Laufzeit und nicht bereits zur Compilezeit).

Da die Laufzeit die `catch`-Blöcke bei der Suche nach dem richtigen Block strikt von oben nach unten durchgeht, ist es wichtig die Blöcke vom Spezifischen zum Allgemeinen hin

anzuordnen. D. h. man muss zuerst den `catch`-Block mit der Ausnahme schreiben, die in der Vererbungshierarchie am weitesten *unten* steht.

3.5 Beispiel: Mehrere catch-Blöcke [19]




```
Connection connection = getConnection();

try {
    ResultSet rs = readData(connection);

    while (rs.next()) {
        storeToFile(rs.getString(0), rs);
    }
} catch (SQLException ex) {
    // Datenbank-Problem
    ex.printStackTrace();
} catch (FileNotFoundException ex) {
    // Datei nicht gefunden
    System.out.println(ex.getMessage());
} catch (IOException ex) {
    // Allgemeiner IO-Fehler
    System.out.println(ex.getMessage());
}
```

Man erkennt im Beispiel deutlich die Regel, dass die `catch`-Blöcke vom Speziellen zum Allgemeinen hin angeordnet werden müssen. `SQLException` und `IOException` stehen in keiner Beziehung zueinander, sodass hier die Anordnung egal ist. Da `FileNotFoundException` spezifischer als `IOException` ist (siehe Abbildung), muss ihr `catch`-Block jedoch zuerst aufgeführt werden.

3.6 Multi-Catch [21]

- Für einen `catch`-Block können beliebig viele Ausnahmen (durch | getrennt) angegeben werden
- Die Ausnahmen dürfen nicht Subklassen voneinander sein
- Jeder Ausnahme-Typ darf nur einmal genannt werden

```
try {
    openFile();
    writeToDatabase();
}
catch (FileNotFoundException | SQLException e) {
    System.out.println("Ohje, alles kaputt...:" + e);
}
```

Es kommt häufig vor, dass eine Methode viele verschiedene Ausnahmen werfen kann. Dies führt beim Verwender zu einer ganzen Reihe von `catch`-Blöcken, die oft alle dasselbe tun. Eine Möglichkeit wäre natürlich, anstelle vieler `catch`-Blöcke nur einen `catch`-Block der Superklasse `Exception` zu verwenden. Dies gilt aber als sehr schlechter Stil (siehe oben) und ist daher keine befriedigende Lösung.

Seit Java 7 besteht die Möglichkeit, mehrere Ausnahmen in einem einzigen `catch`-Block zu fangen. Die Ausnahmen werden durch `|` getrennt und dürfen in keiner Vererbungsbeziehung zueinander stehen.

Der statische Typ der Referenz auf die Ausnahme (`e` im Beispiel) in einem multi-catch ist dann allerdings `Exception`. Will man auf spezielle Attribute der Ausnahmen zugreifen, muss man einen Cast auf ihren dynamischen Typ durchführen.

3.7 So nicht! [22]

```
public static void printArray(int[] array) {

    try {
        for (int i = 0;;i++) {
            System.out.println(array[i]);
        }
    }
    catch (ArrayIndexOutOfBoundsException ex) {
    }
}
```

Der/die Programmierer:in dieses Beispiels hat versucht, die Prüfung auf die Grenzen des Arrays zu sparen und lieber die `ArrayIndexOutOfBoundsException` abzufangen, da er/sie glaubt, dass das einmalige Fangen einer Ausnahme schneller geht als eine Prüfung bei jedem Schleifendurchlauf. Obwohl dieses Programm semantisch korrekt funktioniert, handelt es sich um einen Missbrauch von Ausnahmen, der zur Folge hat, dass das Programm *viel langsamer* läuft. Der Grund liegt darin, dass die Java-VM normalerweise die Schleifengrenzen nur einmal überprüft und dann sehr effizienten Maschinencode generieren kann. Dies ist hier nicht möglich, da es keine Grenzen im Programm gibt. Deshalb muss sie bei jedem Durchlauf erneut

auf die Grenzen testen und entsprechend eine Ausnahme generieren, wenn sie überschritten wurden.

3.8 Vergleich zu C [23]

```
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
    DieWithError("socket () failed") ;
}

if (bind(servSock, (struct sockaddr *)&echoServAddr,
       sizeof(echoServAddr)) < 0) {
   DieWithError ("bind () failed");
}

if (listen(servSock, MAXPENDING) < 0) {
   DieWithError("listen() failed") ;
}
for (;;) {
   if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr,
          &clntLen)) < 0) {
      DieWithError("accept() failed");
   }
}
```

Hier ist noch einmal ein Beispiel der Fehlerbehandlung in der Programmiersprache C dargestellt, das im ersten Kapitel bereits gezeigt wurde.

```
try {
   ServerSocket servSocket = new ServerSocket();
   servSocket.bind(new InetSocketAddress(8080));

   for (;;) {
      Socket clntSocket = servSocket.accept();
      // ...
   }
}
catch (SocketTimeoutException e) {
   // Fehlerbehandlung
}
catch (IOException e) {
   // Fehlerbehandlung
}
```

Bei der Java-Variante des Programms kann man sehr gut die Unterschiede erkennen. Hier vermischt sich die Fehlerbehandlung nicht mehr mit dem normalen Programmablauf. Man kann

deutlich sehen, wie man einen Netzwerkserver programmiert, ohne von der Fehlerbehandlung verwirrt zu werden.

Kapitel 4

Finally

4.1 finally-Block [26]

finally

- Optional kann ein `finally`-Block angegeben werden
- Dient Aufräumarbeiten
  - Schließen von Datenbankverbindungen und Dateien
  - Freigabe von Ressourcen, etc.
- Wird immer ausgeführt, egal wie `try`-Block verlassen wurde
  - durch normalen Ablauf
  - durch explizites werfen einer Ausnahme
  - durch fangen und behandeln einer Ausnahme
  - durch eine Ausnahme, die nicht gefangen wird
  - durch ein `return`-Statement

Es kommt sehr häufig vor, dass man bestimmte Aktionen vorbereitet (z. B. eine Datei öffnet), dann eine Aktion durchführt (in die Datei schreibt) und dann Aufräumarbeiten durchführen muss (die Datei schließen). Wenn jetzt während der Durchführung ein Fehler auftreten kann, wird man entsprechende `try-catch`-Block benutzen. Tritt ein Fehler auf, so muss man in den meisten Fällen trotzdem die Aufräumarbeiten durchführen. Würde man diese im `try`-Block machen, würden sie im Fehlerfall nicht ausgeführt. Macht man sie im `catch`-Block, werden sie nur im Fehlerfall ausgeführt. Es fehlt also die Möglichkeit, Aufräumarbeiten sowohl im Fehler- als auch im Nicht-Fehler-Fall durchzuführen.

Genau zu diesem Zweck dient der `finally`-Block, der durch das Schlüsselwort `finally` eingeleitet wird. Die Anweisungen in diesem Block werden auf jeden Fall ausgeführt, egal, ob ein Fehler auftritt oder nicht. Sogar wenn der `try`- oder `catch`-Block mit `return` verlassen werden sollte, wird vorher noch der `finally`-Block ausgeführt, bevor die Methode zurückkehrt.

`finally` kann auch ohne `catch` benutzt werden

```
Connection connection = getConnection();

try {
```

```
    ResultSet rs = readData(connection);

    while (rs.next()) {
        // Daten bearbeiten
    }

    return;
}
finally {
    releaseConnection(connection);
}
```

4.2 Beispiel: finally-Block [27]



```
Connection connection = getConnection();

try {
    ResultSet rs = readData(connection);

    while (rs.next()) {
        // Daten bearbeiten
    }

    return;
} catch (SQLException ex) {
    // Datenbank Problem behandeln
} finally {
    releaseConnection(connection);
}
```

In diesem Beispiel garantiert der `finally`-Block, dass die Methode `releaseConnection()` auf jeden Fall durchgeführt wird, auch wenn der `try`-Block durch einen Fehler abrupt beendet werden sollte.

4.3 Ausnahmen im finally-Block [28]



```
Connection connection = getConnection();

try {
    ResultSet rs = readData(connection);

    while (rs.next()) {
        // Daten bearbeiten
    }
} catch (SQLException ex) {
```

```
    // Datenbank Problem behandeln
} finally {
    try {
        releaseConnection(connection);
    }
    catch (SQLException ex) {
        // Hier kann man nichts mehr machen
    }
}
```

Da der `finally`-Block für Aufräumarbeiten gedacht ist, sollte man besonders sorgfältig auf den Fall achten, dass er selbst durch eine Exception beendet wird. Normalerweise kann man Ausnahmen, die beim Aufräumen entstehen nicht mehr adäquat behandeln. Deshalb findet man häufig die Konstruktion, dass in einem `finally`-Block noch einmal ein `try-catch`-Block geschachtelt ist, der die Ausnahmen beim Aufräumen fängt. Der `catch`-Block ist in diesem Fall häufig leer, was an anderer Stelle unerwünscht, im `finally`-Block aber normal ist.

Auf jeden Fall sollte man verhindern, dass der `finally`-Block durch eine Ausnahme oder ein `return` abrupt beendet wird. Das folgende Beispiel zeigt warum.

4.4 Abruptes Beenden des finally-Blocks [29]

```
public class Finally {

    public static boolean f() {
        try {
            return true;
        } finally {
            return false;
        }
    }

    public static void main(String[] args) {
        System.out.println(f());
    }
}
```

```
false
```

Ein abruptes Beenden des `finally`-Blocks ist normalerweise nicht erwünscht und gilt als Programmierfehler. Im Beispiel kann man sehen, was passiert, wenn er trotzdem durch ein `return` abrupt beendet wird: der vom `finally`-Block zurückgegebene Wert wird letztendlich aus der Methode zurückgeliefert und nicht der vom `try`-Block. Das `return` im `try`-Block wird

zwar ausgeführt, hat aber keine Wirkung. Eventuelle Seiteneffekte in diesem würden aber auftreten.

4.5 try-with-resources Statement [31]

- Im Kopf eines try/catch-Blockes können Ressourcen (z. B. Streams oder Datenbank-Verbindungen) angefordert werden
- Diese werden automatisch beim Verlassen Blocks geschlossen, ohne dass man dies explizit im finally-Block machen müsste
- Basis ist ein neues Interface AutoCloseable, das die Resource implementieren muss

```
try(FileInputStream fis = new FileInputStream("/tmp/file1");
   FileOutputStream fos = new FileOutputStream("/tmp/file2")) {

    int i = fis.read();
    fos.write(i);
}
catch (IOException e) {
    // Ausnahme behandeln
}
```

Eines der lästigen Probleme beim Umgang mit Dateien und Netzwerkverbindungen ist, dass man sie korrekt schließen muss und zwar auch dann, wenn ein Fehler aufgetreten ist. Damit man sie aber in einem finally-Block schließen kann, muss man die Referenzen vor dem try-Block deklarieren, da sie ansonsten im finally-Block nicht sichtbar sind. Hinzu kommt noch, dass finally innerhalb der VM für einen erheblichen Implementierungsaufwand sorgt, sodass man seitens der Java-VM-Entwickler gerne finally sowenig wie möglich nutzen möchte.

try-with-resources Statement

Das mit Java 7 eingeführte **try-with-resources Statement** erlaubt es, Ressourcen, die am Ende eines try-catch-Blocks geschlossen werden sollen, einfach im Kopf des Blocks anzugeben. Der Java-Compiler kümmert sich dann darum, dass sie korrekt geschlossen werden und zwar unabhängig davon, ob ein Fehler aufgetreten ist oder nicht.

Ohne die Erweiterung hätte man obiges Beispiel wie folgt programmieren müssen:

```
FileInputStream fis = new FileInputStream("/tmp/file1");
FileOutputStream fos = new FileOutputStream("/tmp/file2");

try {
    int i = fis.read();
    fos.write(i);
}
catch (IOException e) {
    // Ausnahme behandeln
}
finally {
    try {
```

```
        fis.close();
    }
    catch (IOException e) { /* ignore */ }

    try {
        fos.close();
    }
    catch (IOException e) { /* ignore */ }
}
```

Kapitel 5

Call-Stack

5.1 Der Call-Stack-Mechanismus [33]

Aufrufhierarchie

- Ausnahmen werden entlang der **Aufrufhierarchie** (call stack) propagiert
- Wenn eine Ausnahme nicht von einem `try-catch`-Block behandelt wird, wird sie an den Aufrufer weitergegeben
- Gelangt die Ausnahme bis zur main-Methode und wird dort nicht behandelt, wird das Programm abgebrochen
- Hierdurch kann die Ausnahme bis zu demjenigen wandern, der sie beheben kann

5.2 Beispiel: Call-Stack-Mechanismus [34]

```
Exception in thread "main" java.io.FileNotFoundException:
          /tmp/input (No such file or directory)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:106)
    at java.io.FileInputStream.<init>(FileInputStream.java:66)
    at pr2.CallStack.aggregate(CallStack.java:26)
    at pr2.CallStack.writeData(CallStack.java:17)
    at pr2.CallStack.mergeData(CallStack.java:22)
    at pr2.CallStack.main(CallStack.java:9)
```

Call-Stack-Mechanismus

Das Ausnahmen in einer Methode gefangen werden können, wurde bereits gezeigt. Was passiert wenn sie nicht gefangen werden? Hierfür gibt es in Java einen Mechanismus (**Call-Stack-Mechanismus**), der dafür sorgt, dass die Ausnahme an den Aufrufer der Methode weitergeleitet wird. Behandelt dieser die Ausnahme ebenfalls nicht, wird sie an dessen Aufrufer weitergegeben etc.

Behandelt keine Methode in der Aufrufhierarchie die Ausnahme, so kommt sie letztendlich immer in der main-Methode an. Wird sie auch dort nicht behandelt, wird das Programm abgebrochen und der Fehler wird auf der Console ausgegeben.

```
public static void main(String[] args) throws IOException {
    mergeData("/tmp/input", "/tmp/output");
}

public static void mergeData(String inFile, String outFile) throws IOException {
    writeData(inFile, outFile);
}

public static void writeData(String inFile, String outFile) throws IOException {
    aggregate(5, 5, inFile);
}

public static int aggregate(int a, int b, String inFile) throws IOException {
    FileInputStream f = new FileInputStream(inFile);
    return 1;
}
```

Wenn man ein Programm betrachtet, in dem Ausnahmen vorkommen können, muss man sich immer Fragen, an welche Stelle die Ursache für die Ausnahme liegt. Dies muss nicht zwingend die Stelle sein, an der die Ausnahme auftritt.

Hier ist eindeutig die Main-Methode für den Fehler verantwortlich, da sie die Dateinamen an die folgenden Methoden übergibt. Insofern ist die `aggregate`-Methode unschuldig an der `FileNotFoundException`, die in ihr entstehen wird, wenn es die Datei `/tmp/input` gar nicht gibt. Sie bekommt den Namen der Datei übergeben und kann nichts dagegen tun, wenn die Datei nicht existiert.



```
public static void main(String[] args) throws IOException {
    mergeData(args[0], "/tmp/output");
}

public static void mergeData(String inFile, String outFile) throws IOException {
    writeData(inFile, outFile);
}

public static void writeData(String inFile, String outFile) throws IOException {
    aggregate(5, 5, inFile);
}

public static int aggregate(int a, int b, String inFile) throws IOException {
    FileInputStream f = new FileInputStream(inFile);
    return 1;
}
```

In diesem Fall ist der/die Benutzer:in für den Fehler verantwortlich, weil er einen ungültigen Dateinamen auf der Konsole angegeben hat.

Eine gute Implementierung der Fehlerbehandlung in der Main-Methode sähe ungefähr wie folgt aus:

```
public static void main(String[] args) {

    if (args.length < 1) {
        System.err.println("Bitte Dateinamen angeben.");
        System.exit(4);
    }


    try {
        mergeData(args[0], "/tmp/output");
    } catch (IOException e) {
        System.err.printf("Datei %s gibt es nicht%n%n", args[0]);
    }
}

public static void mergeData(String inFile, String outFile)
        throws IOException {
    writeData(inFile, outFile);
}

public static void writeData(String inFile, String outFile) throws IOException {
    aggregate(5, 5, inFile);
}

public static int aggregate(int a, int b, String inFile) throws IOException {
    FileInputStream f = new FileInputStream(inFile);
    return 1;
}
```

```
public static void main(String[] args) throws IOException {
    mergeData("/tmp/input", "/tmp/output");
}

public static void mergeData(String inFile, String outFile) throws IOException {
    writeData(outFile);
}

public static void writeData(String outFile) throws IOException {
    aggregate(5, 5);
}

public static int aggregate(int a, int b) throws IOException {
    FileInputStream f = new FileInputStream("/tmp/input");
    return 1;
}
```

In diesem Beispiel liegt die Verantwortung für den Fehler in der `aggregate()`-Methode, da der Dateiname dort festgelegt wird. Sie müsste den Fehler also selbst behandeln und dürfte ihn nicht an die aufrufenden Methoden weitergeben.

Kapitel 6

Arten von Ausnahmen

6.1 Kategorien von Ausnahmen [39]

- **Errors** - Schwerwiegende Fehler, von denen man sich im Allgemeinen nicht erholen kann — Errors
  - Speicher geht aus
  - VM hat andere Probleme
- **(Checked) Exceptions** – Fehler, die behandelt werden sollten — (Checked) Exceptions
  - Datei nicht gefunden
  - Falsche IP-Adresse angegeben
  - Falsches SQL-Statement
- **Runtime Exceptions** – Programmierfehler, die man im Allgemeinen nicht behandelt — Runtime Exceptions
  - Zugriff auf `null`
  - Division durch 0

Es gibt in Java drei grundlegende Arten von Ausnahmen. Sie werden zwar alle mit denselben Mittel erzeugt und behandelt, treten aber in unterschiedlichen Situationen auf.

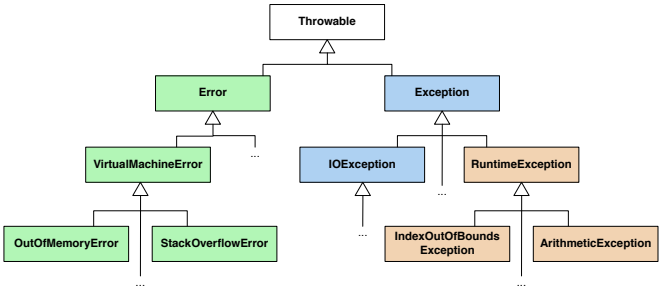
Die JavaDoc der drei Arten von Ausnahmen lautet:

- *Error* - An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The ThreadDeath error, though a „normal“ condition, is also a subclass of Error because most applications should not try to catch it. A method is not required to declare in its throws clause any subclasses of Error that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur.
- *Exception* - The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.
- *RuntimeException* - RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine. A method is not

required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.

- *Throwable* - The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement. Similarly, only this class or one of its subclasses can be the argument type in a catch clause. Instances of two subclasses, java.lang.Error and java.lang.Exception, are conventionally used to indicate that exceptional situations have occurred. Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information (such as stack trace data).

6.2 Hierarchie der Java-Ausnahmen [40]



Errors sind hier grün, Checked Exceptions blau und Runtime Exceptions braun dargestellt.

6.3 Die Klasse Throwable [41]

```
public class Throwable implements Serializable {
    public Throwable() { ... }
    public Throwable(String message) { ... }
    public Throwable(String message, Throwable cause) { ... }
    public Throwable(Throwable cause) { ... }

    public String getMessage() { ... }
    public String getLocalizedMessage() { ... }

    public Throwable getCause() { ... }

    public void printStackTrace() { ... }
    public void printStackTrace(PrintStream s) { ... }
    public void printStackTrace(PrintWriter s) { ... }

    public synchronized native Throwable fillInStackTrace();
    public StackTraceElement[] getStackTrace() { ... }
}
```

Die Klasse `Throwable` ist die Basisklasse aller Ausnahmen in Java. Sie vererbt die wichtigsten Methoden an die von ihr abgeleiteten Ausnahmen. `Throwable`

Die Bedeutung der Methoden ist:

- `public Throwable(String message)`: Anlegen einer neuen Ausnahme mit einer für Menschen lesbaren Nachricht. Die Nachricht wird normalerweise nicht vom Programm ausgewertet (dem reicht der Typ der Ausnahme), sondern ist für menschliche Betrachter gedacht, die einen Fehler analysieren möchten.
- `public Throwable(Throwable cause)`: Zusätzlich zu der Nachricht kann man der Ausnahme noch einen Ursache (`cause`) mitgeben. Hier kann man eine weitere Ausnahme anhängen, die die Ursache für diese Ausnahme ist.
- `public String getMessage()`: Die Nachricht kann mit dieser Methode wieder ausgelesen werden.
- `public String getLocalizedMessage()`: Wie `getMessage()` nur, dass die Nachricht in eine andere Sprache übersetzt werden kann.
- `public Throwable getCause()`: Liefert die Ursache, die im Konstruktor übergeben wurde, wieder zurück.
- `public void printStackTrace()`: Druckt die Aufrufhierarchie aus, die in dem Moment bestand, als die Ausnahme geworfen (nicht gefangen) wurde.
- `public StackTraceElement[] getStackTrace()`: Liefert die Aufrufhierarchie als Array zurück, das man programmatisch untersuchen kann.

6.4 Die Klasse Exception [42]

```
package java.lang;

public class Exception extends Throwable {

    public Exception() { ... }

    public Exception(String message) { ... }

    public Exception(String message, Throwable cause) { ... }

    public Exception(Throwable cause) { ... }

}
```

Die Klasse `Exception` fügt keine neuen Methoden hinzu, sondern enthält nur Konstruktoren, da Konstruktoren in Java nicht vererbt werden. `Exception`

6.5 Beispiele für häufige Ausnahmen [43]

- `ArithmeticException`
- `NullPointerException`
- `NegativeArraySizeException`
- `ArrayIndexOutOfBoundsException`
- `SecurityException`
- `IllegalStateException`
- ...

Welche Bedeutung diese Exceptions haben kann der Leser in der entsprechenden JavaDoc selbst nachschlagen.

6.6 Die handle or declare Regel [44]

handle or declare Regel

Für Ausnahmen gilt die **handle or declare Regel**

- Die Ausnahme wird durch einen try/catch behandelt *oder*
- Die Ausnahme muss bei der Methode mit dem `throws` Schlüsselwort angegeben werden

Gilt nicht für Runtime Exceptions und Error

- Programmierfehler sollen nicht behandelt werden
- Error können (eigentlich) nicht behandelt werden

```
public void storeToFile(String fileName, ResultSet rs)
       throws FileNotFoundException, IOException {
    ...
}
```

Eines der Ziele beim Entwurf der Ausnahmebehandlung für Java war es, dass der Compiler erkennen kann, ob die möglichen Fehler alle behandelt werden. Um dies zu gewährleisten, gibt es die sog. *handle or declare Regel*: Eine Methode muss die Ausnahme entweder mit `catch` (*handle*) fangen oder sie muss in ihrem Methodenkopf angeben, dass sie eben dies nicht tut (*declare*). Für letzteres dient ein neues Schlüsselwort `throws`, das nach der Parameterliste der Methode angegeben wird und alle potenziell von der Methode geworfenen Ausnahmen angibt.

Diese Regel bezieht sich aber nur auf die sogenannten Checked-Exceptions also alle Ausnahmen, die *keine* Runtime-Exceptions und *keine* Errors sind. Die Motivation für diese Beschränkung ist, dass Runtime-Exceptions als Programmierfehler viel zu häufig sind und damit in nahezu jedem Java-Statement auftreten können (man denke an die `NullpointerException`). Außerdem ist es sinnvoll, Programmierfehler zu erkennen und nicht durch Fangen zu verschlucken. Im Gegensatz dazu sind Errors so schwerwiegend, dass ein Fangen nichts nützen würde, da man sie nicht sinnvoll behandeln kann. So gibt es z. B. keine Möglichkeit für ein Java-Programm nach einem `OurOfMemoryError` noch sinnvoll weiterzulaufen.

6.7 Beispiel: Handle or Declare [45]

```
public void openFile(String dateiName) throws IOException, FileNotFoundException {
    // Datei öffnen
}

public void datenSchreiben(String dateiName, String sqlStatement)
        throws FileNotFoundException, IOException, SQLException {
    openFile(dateiName);
    // Mit Datenbank arbeiten
}

public void dateiAnlegen(String dateiName) throws FileNotFoundException, ↩
     IOException {
    try {
        datenSchreiben(dateiName, "SELECT * FROM x");
    } catch (SQLException ex) {
        // Datenbank Problem beheben ;-)
    }
}
```

```
public void userInterface() {

    String dateiName = askUser();

    try {
        dateiAnlegen(dateiName);
    }
    catch (FileNotFoundException ex) {
        // Benutzer:in erneut nach Dateinamen Fragen
    }
    catch (IOException ex) {
        // Benutzer:in auf Problem hinweisen
    }
}
```

Das Beispiel zeigt den Umgang mit der Handle-or-Declare-Regel: die Methode `openFile` kann sowohl an einer fehlenden Datei (`FileNotFoundException`) als auch generellen Problemen beim Dateizugriff scheitern (`IOException`). Deswegen deklariert die Methode beide Exceptions über `throws`.

Die Methode `datenSchreiben` verwendet die `openFile`-Methode. Damit können potenziell ebenfalls die beiden Ausnahmen `FileNotFoundException` und `IOException` entstehen. Da sie keine von beiden fängt, muss sie beide weiter deklarieren. Zusätzlich führt Sie noch einen Datenbankzugriff durch, der einen Fehler durch eine `SQLException` anzeigen könnte (der Datenbankzugriff ist hier nur durch den Kommentar angedeutet.)

Die Methode `dateiAnlegen` behandelt die `SQLException`, nicht jedoch die `IOException` oder die `FileNotFoundException`, weswegen sie beide per `throws` deklarieren muss.

Die letzte Methode `userInterface` fragt den Dateinamen vom Benutzer ab und behandelt alle noch vorhandenen Ausnahmen selbst. Deshalb muss sie keinen Ausnahme per `throws` deklarieren.

6.8 Ausnahmen und Überschreiben [47]

Überschriebene Methoden können

- Deklarierte Ausnahmen unverändert lassen *oder*
- Deklarierte Ausnahmen weglassen *oder*
- Deklarierte Ausnahmen durch ihre Subklassen ersetzen *oder*
- Ausnahmen hinzufügen, die Subklassen der deklarierten sind

Sie können auf keinen Fall neue Checked Exceptions hinzufügen

Grund: Polymorphie

überschriebene Methoden

In einem vorherigen Kapitel wurde bereits diskutiert, dass **überschriebene Methoden** die Sichtbarkeit und den Rückgabetyp nicht ändern dürfen. Im Zusammenhang mit Ausnahmen muss diese Regel noch erweitert werden: Überschriebene Methoden dürfen mit `throws` deklarierte Ausnahmen unverändert lassen, Ausnahmen weglassen oder durch Subklassen ersetzen.

Der Hintergrund liegt in der Polymorphie begründet. Der statische Typ der Referenzvariable legt fest, welche Methoden über die Variable aufgerufen werden können. Damit wird aber über den statischen Typ bestimmt, welche Ausnahmen potenziell aus einem Methodenaufrufe entstehen können. Da der Verwender einer Variable nur deren statischen Typ kennt, wird er sich bei der Fehlerbehandlung an eben diesem orientieren. Würde man jetzt gestatten, dass eine überschriebene Methode in einer Subklasse neue Ausnahmen werfen dürfte, hätte der Verwender keine Chance sich darauf einzurichten. Daher wird dies vom Compiler unterbunden. Anders ausgedrückt kann man sagen, dass die `catch`-Blöcke, die für einen Methodenaufruf auf einem Typ geschrieben wurden, genauso für alle Subtypen passen müssen.

6.9 Beispiel: Überschreiben und Exceptions [48]

```
class A {
    public void method() throws IOException { ... }
}
class B extends A {
    public void method() throws IOException, FileNotFoundException { ... }
}
class C extends A {
    public void method() throws FileNotFoundException { ... }
}
class D extends A {
```

```
    public void method() { ... }
}
class E extends A {
    public void method() throws SQLException { ... } // FEHLER!!
}
```

Nach den gerade erläuterten Regel ist die überschriebene Methode `method` in der Klasse `E` unzulässig, weil sie mehr Ausnahmen wirft, als die gleichnamige Methode in `A`. Ein Verwender würde überrascht, denn er würde nur `catch`-Blöcke für `IOException` vorsehen.

Die Klassen `B`–`D` sind zulässig:

- `B`: Da `FileNotFoundException` eine Subklasse von `IOException` ist, ist diese zusätzliche Ausnahme zulässig, da ein `catch`-Block für `IOException` sie auch fangen würde.
- `C`: `FileNotFoundException` ist eine Subklasse von `IOException` und würde daher auch von vorhandenen `catch`-Blöcken erfasst.
- `D`: Über das Weglassen einer Ausnahme wird kein Verwender böse sein. Der vorhandene `catch`-Block wird nicht genutzt, es entsteht aber kein Schaden.

6.10 Ausnahmen und Programmierstil [49]

Folgende Konstruktionen sollten vermieden werden

- *Sehr lange `try`-Blöcke* – Der `try`-Block sollte nur den Code umfassen, der den Fehler erzeugen kann
- *Catchen von Exception* – Da `Exception` die Mutter aller Ausnahmen ist fängt man damit auch Runtime-Exceptions, die man aber nicht fangen sollte
- *Leere `catch`-Blöcke* – Eine Ausnahme signalisiert einen Fehlerfall, sie einfach zu ignorieren ist fast nie nichtig
- *printStackTrace()* – `catch`-Blöcke, die die Ausnahme nur ausdrucken, ignorieren ebenfalls den Fehler statt ihn zu behandeln

Diese Regeln gelten auch für die Pflichtübungen. Verletzungen führen grundsätzlich zu Punktabzug.

6.11 Beispiel: Schlechter Stil [50]

```
try {
    FileInputStream fis = new FileInputStream("/tmp/gibtsnicht");
    String s = null;
    s = s.toUpperCase();
} catch (Exception e) { // SCHLECHT!!
    // Fehlerbehandlung
}
```

```
FileInputStream fis = null;

try {
   fis = new FileInputStream("/tmp/gibtsnicht");
} catch (IOException e) {
    e.printStackTrace(); // SCHLECHT!!
}

fis.read();
```

Das Beispiel zeigt einen schlechten Stil im Umgang mit Ausnahmen, der zu echten Problemen führen wird.

Das Fangen von `Exception` im ersten `catch`-Block führt dazu, dass nicht nur eine `FileNotFoundException` aus dem Dateizugriff gefangen wird, sondern alle Runtime-Exceptions. Hierdurch wird der Programmierfehler, der sich noch im ersten `try`-Block befindet, verschleiert. Der/die Programmierer:in merkt gar nicht, dass eine `NullPointerException` auftritt, da diese in der Fehlerbehandlung für den Dateizugriff untergeht. Umgekehrt kann es auch passieren, dass man einen Fehler bei der Datei sucht obwohl diese ganz normal gelesen werden könnte und der `catch`-Block nur wegen der `null`-Referenz angesprungen wurde.

Im zweiten `catch`-Block wird zwar die richtige Ausnahme gefangen, diese wird aber einfach auf der Konsole ausgedruckt, anstatt sie zu behandeln. Da echte Programme oft unbeaufsichtigt laufen, ist die Verwendung der Konsole grundsätzlich eine schlechte Idee.

Kapitel 7

Eigene Ausnahmen

7.1 Ausnahmen selbst schreiben [52]

- Man legt eine Subklasse von `Exception` an
- Man gibt ihr einen Namen, der auf „Exception“ endet
- Man fügt Daten hinzu, die für die Ausnahme wichtig sind
- Man bietet passende Konstruktoren zu den Daten an, mindestens aber zwei Konstruktoren:
  - `MyException()`
  - `MyException(String message)`

Man kann nicht nur eigene Exceptions schreiben, man sollte es auch. Zu einem guten API gehören auch passende Ausnahmen. Allerdings sollte man dies nicht übertreiben und so häufig wie möglich auch vorhandene Exceptions der Java-Klassenbibliothek wieder verwenden, da man hier dem Verwender des APIs bereits bekannte Ausnahmen wie z. B. `IllegalArgumentException` etc. anbieten kann.

Da bereits die Klasse `Throwable` das `Serializable` Interface implementiert (Details zu Serialisierung kommen später) muss man dafür sorgen, dass die selbstgeschriebenen Ausnahmen serialisierbar sind. Aus diesem Grund warnt Eclipse beim Anlegen einer neuen Exception, man solle eine `serialVersionUID` vergeben. Diese Warnung können wir an dieser Stelle noch nicht diskutieren.

7.2 Ausnahmen selbst werfen [53]

- Nur Subklassen von `Throwable` können geworfen werden
- Im Allgemeinen wird man nur Runtime Exceptions und (Checked) Exceptions werfen
- Errors sind der VM vorbehalten
- Ausnahmen werden mit dem Schlüsselwort `throw` geworfen
- Ausnahmen sollten frisch mit `new` erzeugt werden
- Syntax: `throw new WhatEverException(...)`

Da Ausnahmen ganz normale Java-Objekte sind, muss man nur eine entsprechende Klasse schreiben, um eine eigene Art von Ausnahmen zu definieren. Die Ausnahme-Objekte werden, wie alle Objekte, mit `new` erzeugt.

Damit hat man aber erst ein Objekt erzeugt, es aber noch nicht auf den Weg gebracht. Hierzu muss man es mit dem Schlüsselwort **throw** werfen (**Werfen von Ausnahmen**).

throw
Werfen von Ausnahmen

Da Errors für die VM reserviert sind, sollten Sie ausschließlich Runtime- und Checked-Exceptions selbst schreiben und werfen. D. h., dass Sie nicht direkt von `Throwable` ableiten, sondern immer von `Exception` oder `RuntimeException`.

7.3 Beispiel: Exceptions werfen [54]

```
try {
    System.out.println("A");

    if (true) {
        throw new IOException("Festplatte abgebrannt");
    }

    System.out.println("B");
}
catch (IOException e) {
    System.out.println("Fehler: " + e.getMessage());
}
```



```
A
Fehler: Festplatte abgebrannt
```



Das `if (true)` ist an dieser Stelle tatsächlich nötig, weil der Compiler sonst die Zeile `System.out.println("B")` als nicht erreichbaren Code (unreachable Code) gemeldet hätte.

Das Beispiel zeigt, dass man eine Ausnahme direkt nach dem Werfen mit `throw` wieder einfangen kann.

```
public void method() {
    try {
        System.out.println("A");
        werfer();
        System.out.println("B");
    }
    catch (IOException e) {
        System.out.println("Fehler: " + e.getMessage());
    }
}
```



```
private void werfer() throws IOException {
    throw new IOException("Festplatte explodiert");
}
```

```
A
Fehler: Festplatte explodiert
```

Im Beispiel ist eine Methode `werfer` zu sehen, die mit `throw` eine Ausnahme vom Typ `IOException` wirft und auch korrekt über `throws` deklariert. Die Methode `method` ruft `werfer` auf und fängt die Exception sofort mit einem `catch`. Die Ausgabe des Programms ist damit „A“ und dann die Nachricht aus der Ausnahme. Zur Ausgabe „B“ kommt es nicht mehr, weil vorher die Ausnahme auftritt und der `try`-Block abrupt beendet wird.

7.4 Beispiel: Die Exception [58]

```
public class ServerException extends Exception {
    private int port;
    private String host;

    public ServerException(String message, String host, int port) {
        super(message);
        this.host = host;
        this.port = port;
    }

    public int getPort() {
        return port;
    }

    public String getHost() {
        return host;
    }
}
```

Man kann nicht nur vorhandene Ausnahmen (z. B. `IOException`) werfen, sondern auch eigene Ausnahmen schreiben. Das Beispiel zeigt eine selbstgeschriebene Ausnahme mit dem Namen `ServerException`. Diese enthält weitere Daten zum Problem, das die Ausnahme ausgelöst hat. Hier sind dies Informationen zu einer Netzwerkverbindung, die nicht aufgebaut werden konnte. `port` und `host` bezeichnen einen Server im Internet eindeutig.

Auf die übliche Vorgehensweise, bei einer Exception mindestens einen Default-Konstruktor und einen mit einer Nachricht anzubieten, wurde hier aus Platzgründen verzichtet. Folgende Konstruktoren sollten noch zur `ServerException` hinzugefügt werden:

- `ServerException()`
- `ServerException(String message)`
- `ServerException(Throwable cause)`
- `ServerException(String message, Throwable cause)`

7.5 Beispiel: Der Werfer [59]

```
public class Server {
    public static void connect(String host, int port)
            throws ServerException {

        int result = connectInternal(host, port);

        if (result == -1) {
            throw new ServerException("Cannot connect to "
                + host + ":" + port, host, port);
        }

        // ...

    }

    private static native int connectInternal(String host, int port);
}
```

Die Klasse `Server` symbolisiert eine Komponente, die Netzwerkverbindungen aufbauen kann. Die eigentliche Verbindung wird von der Methode `connectInternal` hergestellt, die nativ, d. h. nicht in Java, sondern in C implementiert ist. Aus diesem Grund kann `connectInternal` keine Ausnahmen werfen, sondern zeigt ein Verbindungsproblem durch einen Rückgabewert von –1 an.

Die Methode `connect` ruft die interne Methode auf und wirft im Fehlerfall eine Ausnahme vom Typ `ServerException`. Neben der menschen-lesbaren Informationen „Cannot connect to“ werden noch die entsprechenden Attribute `host` und `port` der Ausnahme gesetzt.

7.6 Beispiel: Der Fänger [60]

```
public class Client {
    public void doIt() {

        try {
            Server.connect("server1", 22);
        } catch (ServerException e1) {
            System.err.println("Keine Verbindung zum Server "
                    + e1.getHost() + " auf Port " + e1.getPort());
```

```
            try {
                Server.connect("server2", 22);
            } catch (ServerException e2) {
                System.err.println("Ersatzserver geht auch nicht: "
                        + e2.getHost() + " auf Port " + e2.getPort());
            }
        }
    }
}
```

Die Klasse `Client` versucht eine Verbindung mithilfe der Klasse `Server` aufzubauen. Gelingt dies nicht, wird ein Ersatzserver versucht.

Man erkennt an diesem Beispiel zwei Dinge:

- try-catch-Blöcke können ineinander geschachtelt werden
- Informationen aus der Ausnahme können ausgewertet werden. Hier wird insbesondere nicht die englische Nachricht aus der Ausnahme verwendet, sondern `Client` übersetzt die Meldung unter Verwendung der Attribute der Ausnahme.

Kapitel 8

Ausnahmen und JUnit

8.1 Wie teste ich auf Ausnahmen? [62]

Es gibt grundsätzlich zwei Möglichkeiten auf das Auftreten einer Ausnahme zu testen

- Man macht ein `try { ... } catch { ... }` und nutzt `fail()` bzw. `assertTrue(true)`
- Man macht eine eigene Testmethode und annotiert sie mit `Assertions.assertThrows(Ausnahme.class, ...`

Wichtig ist, den Fall zu berücksichtigen, dass die Ausnahme nicht geworfen wird

8.2 Beispiel: try und catch [63]

```
public class TestMitTryCatch {

    @Test
    public void testAufAusnahme() {

        try {
            new String((String) null);
            fail();

        }
        catch (NullPointerException ex) {
            assertTrue(true);
        }
    }
}
```

Das Beispiel zeigt, wie man in einem JUnit-Test feststellen kann, ob eine Methode eine erwartete Ausnahme wirft. Wichtig ist hier das `fail()` nach der String-Operation. Würde dieses fehlen, dann würde der Test nicht korrekt anzeigen, ob die Ausnahme geworfen wird oder nicht. Fehlt das `fail()`, würde die Methode, wenn keine Ausnahme geworfen wird, einfach zurückkehren und JUnit würde den Test als erfolgreich zählen.

8.3 Beispiel: assertThrows [64]

```
public class TestMitAssertThrows {

    @Test
    public void testAufAusnahme() {
        Assertions.assertThrows(NullPointerException.class, () -> {
            new String((String) null);
        });
    }
}
```



Deutlich einfacher kann man auf Ausnahmen testen, wenn man JUnit über `assertThrows` mitteilt, dass eine bestimmte Ausnahme erwartet wird. Durch diesen Ansatz spart man viel Schreibarbeit, was man sofort sieht, wenn man dieses Beispiel mit dem äquivalenten vorhergehenden vergleicht.

assertThrows

Kapitel 9

Quiz

9.1 Quiz: Ausnahmen und Überschreiben [66]

```
protected void methode() throws IOException { /* irrelevant */ }

protected void methode() throws IOException
protected void methode() throws FileNotFoundException, IOException
protected void methode() throws FileNotFoundException
protected void methode() throws Exception
protected void methode()
public void methode() throws FileNotFoundException, IOException
public void methode() throws FileNotFoundException
public void methode() throws Exception
public void methode()
private void methode() throws FileNotFoundException, IOException
private void methode() throws FileNotFoundException
private void methode() throws Exception
public void methode() throws ArrayIndexOutOfBoundsException
private void methode() throws ArrayIndexOutOfBoundsException
protected void methode() throws ArrayIndexOutOfBoundsException
protected int methode() throws IOException
```



```
protected void methode() throws IOException { /* irrelevant */ }

protected void methode() throws IOException
protected void methode() throws FileNotFoundException, IOException
protected void methode() throws FileNotFoundException
protected void methode() throws Exception // FALSCH!!
protected void methode()
public void methode() throws FileNotFoundException, IOException
public void methode() throws FileNotFoundException
public void methode() throws Exception // FALSCH!!
public void methode()
private void methode() throws FileNotFoundException, IOException // FALSCH!!
private void methode() throws FileNotFoundException // FALSCH!!
private void methode() throws Exception // FALSCH!!
```



```
public void methode() throws ArrayIndexOutOfBoundsException
private void methode() throws ArrayIndexOutOfBoundsException // FALSCH!!
protected void methode() throws ArrayIndexOutOfBoundsException
protected int methode() throws IOException // FALSCH!!
```

Ausnahmen dürfen beim Überschreiben von Methoden nicht erweitert werden, auch dürfen keine neuen (checked) Ausnahmen hinzugefügt werden. Weiterhin darf beim Überschreiben der Rückgabetyp der Methode nicht verändert werden oder deren Sichtbarkeit eingeschränkt. Runtime-Exceptions dürfen jedoch jederzeit hinzugefügt werden, da sie nicht unter die Handle-or-Declare-Regel von Java fallen. Deswegen kann man die `ArrayIndexOutOfBoundsException` ohne Probleme hinzufügen.

9.2 Quiz: finally [68]



```
try {
    try {
        System.out.print("A");
        thrower();
        System.out.print("B");
        return;
    } catch (FileNotFoundException ex) {
        System.out.print("C");
    } finally {
        System.out.print("D");
    }
    System.out.print("E");
    return;
} finally {
    System.out.print("F");
}
// ACDEF
```

Was passiert, wenn `thrower()` eine `FileNotFoundException` wirft?

Das Programm gibt „A“ aus, dann wird von der Methode `thrower()` einen `FileNotFoundException` geworfen, mit der Folge, dass der entsprechende `try`-Block abrupt beendet wird. Die Ausführung springt dann in den entsprechenden `catch`-Block für die Ausnahme, in dem „C“ ausgegeben wird. Danach wird der `finally`-Block ausgeführt („D“) und die Ausführung geht nach dem ersten `try-catch`-Block weiter („E“). Trotz des return-Statements wird noch der letzte `finally`-Block ausgeführt („F“).

Index

(Checked) Exceptions, 21

abrupt beendet, 6
Aufrufhierarchie, 17
Ausnahmen, 6

Call-Stack-Mechanismus, 17
catch-Blöcke, 6

Errors, 21
Exception, 6

Fangen einer Ausnahme, 6
Fehlerbehandlungsroutine, 6

handle or declare Regel, 24

normal beendet, 6

Runtime Exceptions, 21

throw, 30
try-Block, 6
try-with-resources Statement, 15

Werfen von Ausnahmen, 30

überschriebene Methoden, 26

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Input und Output



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**
- 1.1 Fragen [5] . . . 1
- 1.2 Begriffe [6] . . . 1
- 1.3 Links zur Java-Insel [7] . . . 1

2 Einführung **2**
- 2.1 Motivation [9] . . . 2
- 2.2 Beispiel: Datei in C öffnen [10] . . . 2
- 2.3 Eingabe/Ausgabe in Java [11] . . . 3

3 Streams **4**
- 3.1 Klassenhierarchie: Input Streams [13] . . . 4
- 3.2 Die Methoden von InputStream [15] . . . 5
- 3.3 Bytes von einem Stream lesen [17] . . . 5
- 3.4 Blöcke von einem Stream lesen [19] . . . 6
- 3.5 Klassenhierarchie: Output Streams [20] . . . 7
- 3.6 Die Methoden von OutputStream [22] . . . 7
- 3.7 Bytes auf einen Stream schreiben [23] . . . 8
- 3.8 Blöcke auf einen Stream schreiben [24] . . . 8
- 3.9 FileInputStream [25] . . . 8
- 3.10 FileOutputStream [26] . . . 9
- 3.11 Beispiel: Anhängen an Stream [27] . . . 9
- 3.12 Beispiel: Einfaches Kopierprogramm [28] . . . 10
- 3.13 Beispiel: Naive Prüfsummenbestimmung [29] . . . 10
- 3.14 Filter Streams und Decorator Pattern [30] . . . 11
- 3.15 Beispiel: Buffering [31] . . . 11
- 3.16 DataOutputStream / DataInputStream [32] . . . 12
- 3.17 Methoden von DataOutputStream [33] . . . 12
- 3.18 Beispiel: DataOutputStream [35] . . . 12
- 3.19 Methoden von DataInputStream [36] . . . 13
- 3.20 Beispiel: DataInputStream [38] . . . 13

4 Reader / Writer **15**
- 4.1 Klassenhierarchie: Reader [40] . . . 15
- 4.2 Methoden von Reader [42] . . . 15

4.3 Zeichen von einem Reader lesen [44] . . . 16
4.4 Blöcke von einem Reader lesen [46] . . . 17
4.5 Beispiel: StringReader [47] . . . 17
4.6 BufferedReader [48] . . . 18
4.7 Beispiel: BufferedReader [49] . . . 18
4.8 Beispiel: Eigener FilterReader [50] . . . 18
4.9 Klassenhierarchie: Writer [52] . . . 19
4.10 Die Methoden von Writer [54] . . . 20
4.11 Zeichen in einen Writer schreiben [55] . . . 20
4.12 Strings in einen Writer schreiben [56] . . . 21
4.13 BufferedWriter [57] . . . 21
4.14 Beispiel: CharArrayWriter [58] . . . 21
4.15 InputStreamReader etc. [59] . . . 22
4.16 Beispiel: Einfacher HTTP-Client [60] . . . 22
4.17 Die Klasse PrintWriter [62] . . . 23

5 Konsolen IO 24
5.1 Konsole IO [64] . . . 24
5.2 Beispiel: Echo [65] . . . 24
5.3 Beispiel: Umleitung von System.out [66] . . . 25

6 Random Access Files 26
6.1 Der Außenseiter RandomAccessFile [68] . . . 26
6.2 Methoden von RandomAccessFile [69] . . . 26
6.3 Die Modi von RandomAccessFile [71] . . . 27
6.4 Beispiel: RandomAccessFile [72] . . . 27

7 FileSystem und Path 29
7.1 Motivation für Abstraktion [75] . . . 29
7.2 FileSystem und FileSystems [76] . . . 29
7.3 Path [77] . . . 30
7.4 Files [83] . . . 32

8 Serialisierung 34
8.1 Das Problem [86] . . . 34
8.2 Die Lösung: Java Serialisierung [87] . . . 34
8.3 Regeln für die Serialisierung [88] . . . 35
8.4 Beispiel: Serialisierung [89] . . . 36
8.5 Beispiel: Deserialisierung [90] . . . 36
8.6 Objektgeflecht und Serialisierung [91] . . . 37
8.7 Transiente Attribute [92] . . . 37
8.8 Beispiel: Transiente Felder [93] . . . 38
8.9 Eigene Formate für die Serialisierung [95] . . . 39
8.10 Beispiel: readObject() und writeObject() [96] . . . 39
8.11 readResolve() [101] . . . 42
8.12 Beispiel: readResolve() [102] . . . 42

8.13 serialVersionUID [105] 43

9 Ausblick: NIO **45**

9.1 Die Probleme von java.io [107] 45
9.2 Channels und Buffer [108] 45
9.3 Beispiel: Lesen in einen Buffer [109] 46
9.4 Beispiel: Schreiben aus einem Buffer [110] 46
9.5 Buffer [111] 47
9.6 Beispiel: Kopierprogramm [112] 47
9.7 Methoden von Buffer [114] 48
9.8 Weitere Features von NIO [116] 48

Index **iii**

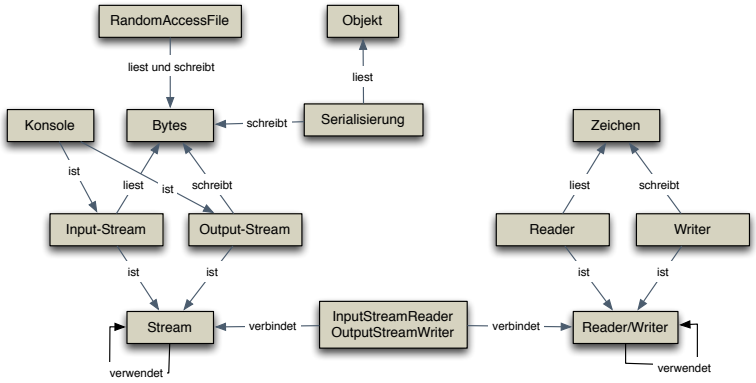
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie komme ich an Daten von der Festplatte?
- Wie kann ich sicher Zeichen verarbeiten, wo es doch so viele unterschiedliche Sprachen und Zeichensätze gibt?
- Wie kann ich Objekte speichern?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 20. Einführung in Dateien und Datenströme

Kapitel 2

Einführung

2.1 Motivation [9]

- In klassischen Programmiersprachen ist **Eingabe/Ausgabe** (input/output) (I/O) Teil der Sprache
- In modernen Programmiersprachen wird IO in Bibliotheken ausgelagert
  - Sockets
  - Pipes
  - Dateien
  - HTTP
  - …
- Häufig sind die Schnittstellen für die verscheiden Datenquellen sehr unterschiedlich

Eingabe/Ausgabe

In Java ist die Ein- und Ausgabe (**I/O**) nicht Teil der Programmiersprache, sondern der Bibliothek. Dies hat den Vorteil, dass man die Bibliotheksfunktionen austauschen kann, was in Java bereits passiert ist: Es gibt neben den normalen I/O-Funktionen, zwei weitere (NIO und NIO2), die für spezielle Aufgabenstellungen besser geeignet sind.

I/O

2.2 Beispiel: Datei in C öffnen [10]

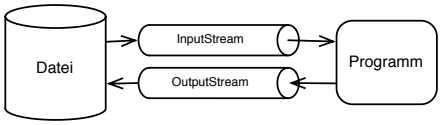
```c
/* fopen example */
#include <stdio.h>
int main ()
{
  FILE * pFile;
  pFile = fopen ("myfile.txt", "w");
  if (pFile!=NULL)
  {
    fputs ("fopen example",pFile);
    fclose (pFile);
  }
  return 0;
}
```

Das Beispiel zeigt, wie man eine Datei in der Programmiersprache C öffnet, den Text „fopen example“ hineinschreibt und danach wieder schließt.

2.3 Eingabe/Ausgabe in Java [11]

Streams

- Java verwendet ein einheitliches Konzept für (nahezu) alle IO-Operationen: **Streams** (Ströme)
- Sie liefern bzw. konsumieren die Daten in sequentieller Form
- Man unterscheidet
  - `Input Streams` – Das Programm liest Bytes von einer Quelle
  - `Output Streams` – Das Programm schreibt Bytes in ein Ziel
  - `Reader` – Wie InputStream nur für Zeichen statt Bytes
  - `Writer` – Wie OutputStream nur für Zeichen statt Bytes



Man wollte in Java vollkommen davon abstrahieren, woher die Daten stammen, die man im Programm verarbeitet. Aus diesem Grund hat man sich für ein Konzept entschieden, das *Streams* heißt: Die Daten werden als Ströme von Bytes oder Zeichen verarbeitet. Die Quellen oder Senken der Daten sind dabei unerheblich und können, ohne Änderungen am Quelltext des Verwenders, geändert werden. So kann man mit denselben Befehlen Daten von der Festplatte, vom Netzwerk oder aus einem anderen Prozess lesen. Die Operationen bleiben gleich, da alle diese Quellen die Daten als Folge von Bytes (oder Zeichen) anliefern können.

Die Streams werden aus der Sicht des Programms benannt:

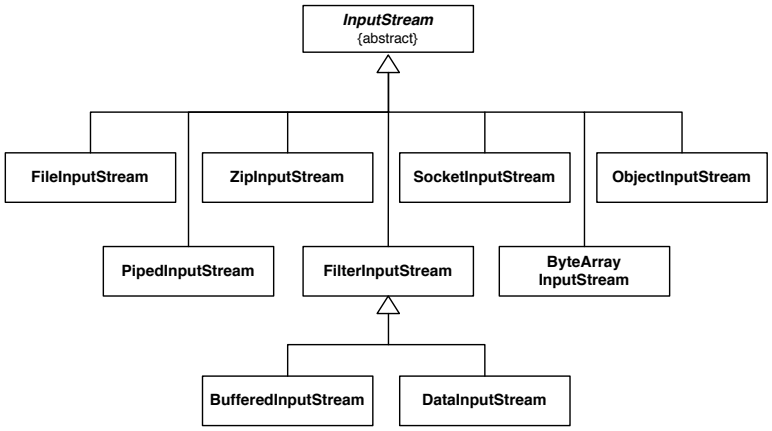
InputStreams
Reader
OutputStreams
Writer

- Daten in das Programm liefern die **InputStreams** und **Reader**.
- Daten aus dem Programm speichern die **OutputStreams** und **Writer**.

Kapitel 3

Streams

3.1 Klassenhierarchie: Input Streams [13]



Man erkennt an der Klassenhierarchie gut den Aufbau. Ausgehend von der Basisklasse `InputStream`, die alle grundlegenden Operationen auf den lesenden Streams festlegt, gibt es für die unterschiedlichen Datenquellen spezielle Subklassen. Diese sind:

`InputStream`

- `FileInputStream` – Dateien — `FileInputStream`
- `ZipInputStream` – Zip-Dateien — `ZipInputStream`
- `SocketInputStream` – Netzwerk-Sockets — `SocketInputStream`
- `ObjectInputStream` – Serialisierung — `ObjectInputStream`
- `PipedInputStream` – Verbindung von Threads — `PipedInputStream`
- `ByteArrayInputStream` – Lesen aus einem Byte-Array — `ByteArrayInputStream`
- `FilterInputStream` – Basisklasse für Filter — `FilterInputStream`
- `BufferedInputStream` – Pufferung — `BufferedInputStream`
- `DataInputStream` – Lesen von Daten — `DataInputStream`

3.2 Die Methoden von InputStream [15]

- `abstract int read()` – liest ein einzelnes Byte vom Stream
- `int read(byte[] b)` – liest vom Stream in das Byte-Array
- `int read(byte[] b, int off, int len)` – liest maximal `len` Bytes in das Byte-Array ab position `off`
- `long skip(long n)` – überspringt n Bytes
- `int available()` – Anzahl der Bytes die voraussichtlich gelesen werden können ohne das die Leseoperation blockiert
- `void close()` – schließt den Stream
- `void mark(int readlimit)` – markiert die aktuelle Position im Stream und vergisst sie erst nach readlimit Bytes die daraufhin gelesen wurden
- `void reset()` – springt zu der mit `mark()` markierten Stelle im Stream zurück
- `boolean markSupported()` – zeigt an, ob der Stream überhaupt das setzen einer Marke mit `mark()` unterstützt

Bezüglich der genauen Arbeitsweise der Methoden und ihrer Dokumentation sei auf die JavaDoc der Klassen verwiesen.

Bemerkenswert ist aber die Tatsache, dass die Methode `read()` ein `int` als Rückgabetyp hat und kein `byte`, obwohl ein `InputStream` die Daten byteweise anliefert. Der größere Datentyp ist nötig, da durch eine -1 signalisiert wird, dass der Stream zu Ende ist. Da aber -1 ein gültiger Byte-Wert ist, hat man hier den größeren Datentyp nehmen müssen. Aus diesem Grund ist es *falsch* den Rückgabewert von `read()` *vor* dem Vergleich mit -1 zu casten. Erst muss mit -1 (als `int`) verglichen werden, dann gecastet, sonst bricht der Lesevorgang möglicherweise zu früh ab, nämlich wenn in den Daten zufällig ein `0xFF` vorkommt, das vorzeichenbehaftet einer -1 entspricht.

3.3 Bytes von einem Stream lesen [17]

```
InputStream fis = new FileInputStream("/tmp/test.txt");

int daten;

while ((daten = fis.read()) != -1) {
    byte b = (byte) daten;
    // jetzt kann man etwas sinnvolles mit den Bytes machen, die aus
    // der Datei gelesen wurden
}

fis.close();
```

In diesem Beispiel wird die -1 richtig behandelt. Erst nach dem Vergleich erfolgt der cast.

```
InputStream fis = new FileInputStream("/tmp/test.txt");

byte b;

while ((b = (byte) fis.read()) != -1) { // FEHLER!!
    // jetzt kann man etwas sinnvolles mit den Bytes machen, die aus
    // der Datei gelesen wurden
}

fis.close();
```

In diesem Beispiel wird die -1 *falsch* behandelt. Es wird erst gecastet und dann verglichen. Der Lesevorgang wird potenziell zu früh beendet, weil -1 ein möglicher Byte-Wert ist und so der Lesevorgang abbricht, wenn dieser Wert in der Quelle auftritt.

3.4 Blöcke von einem Stream lesen [19]

```
InputStream fis = new FileInputStream("/tmp/test.txt");

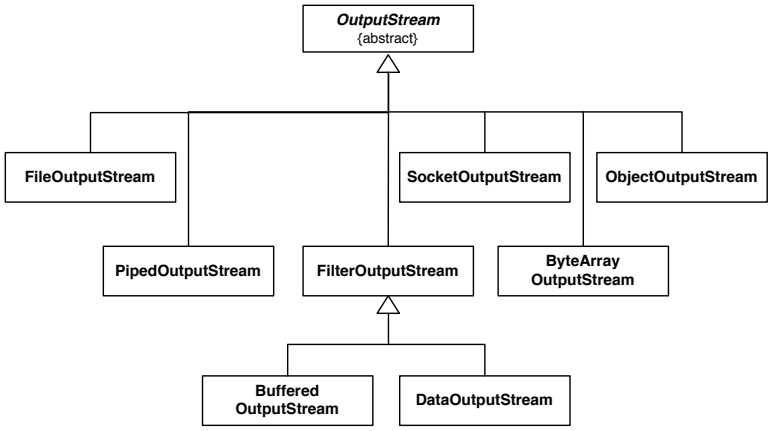
byte[] daten = new byte[1024];
int bytesRead;

while ((bytesRead = fis.read(daten)) > -1) {
    // jetzt kann man etwas sinnvolles mit den Bytes machen, die aus
    // der Datei gelesen wurden
}

fis.close();
```

Das Lesen einzelner Bytes ist ineffizient, daher bietet `InputStream` eine Methode an, mit der man die Bytes blockweise in ein Byte-Array lesen kann. Hier signalisiert der Rückgabewert der Methode, wie viele Bytes gelesen wurden. `-1` zeigt das Ende des Streams an. `0` ist ein korrekter Rückgabewert, da es beim Lesen vom Netzwerk vorkommen kann, dass noch keine neuen Daten eingetroffen sind, die Verbindung aber noch steht. Hier muss man einfach erneut `read` aufrufen.

3.5 Klassenhierarchie: Output Streams [20]



Auch bei den OutputStreams gibt es eine Basisklasse, die das Verhalten festlegt und eine Reihe von konkreten Klassen für die verschiedenen Senken. Diese sind:

- `FileOutputStream` – Dateien — `FileOutputStream`
- `SocketOutputStream` – Netzwerk-Sockets — `SocketOutputStream`
- `ObjectOutputStream` – Serialisierung — `ObjectOutputStream`
- `PipedOutputStream` – Verbindung von Threads — `PipedOutputStream`
- `FilterOutputStream` – Basisklasse für Filter — `FilterOutputStream`
- `ByteArrayOutputStream` – Schreiben in ein Byte-Array — `ByteArrayOutputStream`
- `BufferedOutputStream` – Pufferung — `BufferedOutputStream`
- `DataOutputStream` – Schreiben von Daten — `DataOutputStream`

3.6 Die Methoden von OutputStream [22]

- `abstract void write(int b)` – schreibt ein Byte
- `void write(byte[] b)` – schreibt das gesamte Byte-Array
- `void write(byte[] b, int off, int len)` – schreibt `len` Bytes aus dem Byte-Array beginnend bei `off`
- `void flush()` – leert interne Puffer und bringt die Daten auf die Platte
- `void close()` – schließt den Stream (impliziert ein `flush()`)

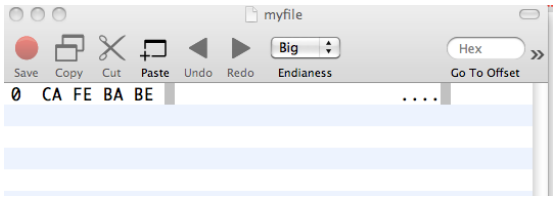
Auch hier sei auf die JavaDoc der Klassen verwiesen. Die Tatsache, dass die `write`-Methode einen `int` als Parameter nimmt, hat keine besondere Bedeutung, sondern dient nur dazu, dass sie symmetrisch zur read-Methode ist. Hierdurch kann man den Wert, der von `read` gelesen wurde direkt `write` übergeben ohne ihn konvertieren zu müssen.

3.7 Bytes auf einen Stream schreiben [23]

```
OutputStream fos = new FileOutputStream("/tmp/myfile");

fos.write(0xca);
fos.write(0xfe);
fos.write(0xba);
fos.write(0xbe);

fos.close();
```

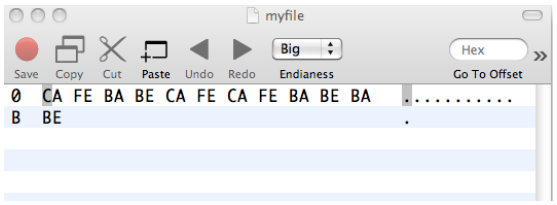


3.8 Blöcke auf einen Stream schreiben [24]

```
OutputStream fos = new FileOutputStream("/tmp/myfile");

byte[] daten = { (byte) 0xca, (byte) 0xfe,
       (byte) 0xba, (byte) 0xbe };

fos.write(daten);
fos.write(daten, 0, 2);
fos.write(daten, 0, 2);
fos.write(daten, 2, 2);
fos.write(daten, 2, 2);
fos.close();
```



3.9 FileInputStream [25]

- FileInputStream

- `public FileInputStream(String name) throws FileNotFoundException`
- `public FileInputStream(File file) throws FileNotFoundException`

`FileInputStream` erbt alle Methoden von `InputStream` und fügt nur zwei Konstruktoren hinzu, um eine Datei zum Öffnen auswählen zu können. Man kann die Datei entweder über den Dateinamen (als `String`) oder ein spezielles `File`-Objekt öffnen (das noch später erläutert wird).

3.10 FileOutputStream [26]

- `FileOutputStream` `FileOutputStream`
  - `public FileOutputStream(String name) throws FileNotFoundException`
  - `public FileOutputStream(String name, boolean append) throws FileNotFoundException`
  - `public FileOutputStream(File file) throws FileNotFoundException`
  - `public FileOutputStream(File file, boolean append) throws FileNotFoundException`

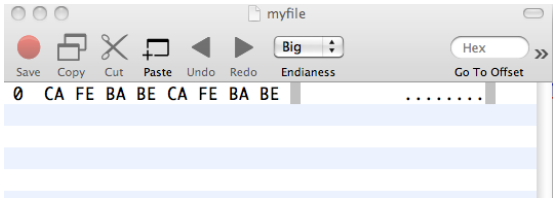
`FileOutputStream` verhält sich analog zum `FileInputStream` nur dass hier über einen weiteren Konstruktorparameter `append` angegeben werden kann, ob die Datei überschrieben (`false`) werden soll oder die geschriebenen Daten hinten an die Datei angehängt werden sollen (`true`).

3.11 Beispiel: Anhängen an Stream [27]

```
byte[] daten = { (byte) 0xca, (byte) 0xfe,
       (byte) 0xba, (byte) 0xbe };

OutputStream fos = new FileOutputStream("/tmp/myfile");
fos.write(daten);
fos.close();

OutputStream fos2 = new FileOutputStream("/tmp/myfile", true);
fos2.write(daten);
fos2.close();
```



3.12 Beispiel: Einfaches Kopierprogramm [28]

```
// Quell und Zieldatei
String quelle = "/tmp/quelle.txt";
String ziel = "/tmp/ziel.txt";

// Streams für Quell und Zieldatei
InputStream in = new FileInputStream(quelle);
OutputStream out = new FileOutputStream(ziel);
byte[] buffer = new byte[1024]; // Puffer für Dateiinhalt
int gelesen;                    // Anzahl der gelesenen Bytes

// Daten aus Quell- in Zieldatei kopieren
while ((gelesen = in.read(buffer)) > -1) {
    out.write(buffer, 0, gelesen);
}

// Streams schließen
in.close();
out.close();
```

3.13 Beispiel: Naive Prüfsummenbestimmung [29]

```
InputStream in = new FileInputStream("/tmp/quelle.txt");
ByteArrayOutputStream out = new ByteArrayOutputStream();

int daten;
while ((daten = in.read()) > -1) {
    out.write(daten);
}

in.close();
out.close();

byte[] bytes = out.toByteArray();
long sum = 0;

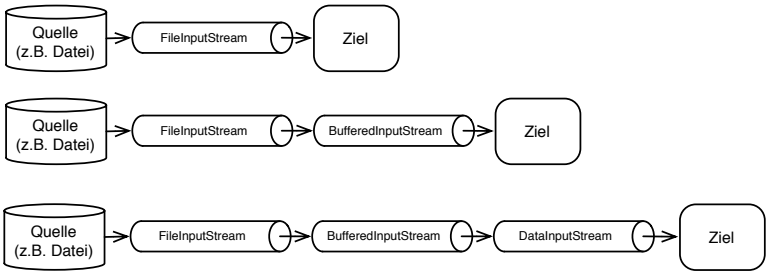
for (int i = 0; i < bytes.length; i++) {
    sum += bytes[i];
```

```
}
System.out.println(sum);
```

3.14 Filter Streams und Decorator Pattern [30]

- Zusätzlich zu den Streams, die direkt mit einer Quelle oder Senke (Datei, Socket etc.) verbunden sind, gibt es die Filter-Streams
- **Filter-Streams** können hinter einen anderen Stream geschaltet werden (**Decorator Pattern**) und können die Daten entsprechend verändern (daher Filter-Streams)

Filter-Streams
Decorator Pattern



Wenn man mehrere Streams miteinander verkettet, fließen die Daten von der Quelle (oder zur Senke) nacheinander durch die Streams. Diese können die Daten dann entsprechend verändern oder umwandeln.

3.15 Beispiel: Buffering [31]

```
String quelle = "/tmp/quelle.txt";
String ziel = "/tmp/ziel.txt";

InputStream in = new BufferedInputStream(
       new FileInputStream(quelle));

OutputStream out = new BufferedOutputStream(
       new FileOutputStream(ziel));

byte[] buffer = new byte[1024];
int gelesen;

while ((gelesen = in.read(buffer)) > -1) {
    out.write(buffer, 0, gelesen);
}

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

Ein häufig eingesetzter Filter-Stream ist der **BufferedInputStream** bzw. der **BufferedOutputStream**. Diese **puffern** (buffer) die Daten und erlauben so eine schnellere Verarbeitung, weil nicht für jede Operation bis auf die Quelle/Senke durchgegriffen werden muss.

BufferedInputStream
BufferedOutputStream
puffern

3.16 DataOutputStream / DataInputStream [32]

- **DataOutputStream** schreibt primitive Datentypen und String in einen anderen Stream und erlaubt so den plattformunabhängigen Datenaustausch
- **DateInputStream** liest die von `DataOutputStream` geschriebenen Daten wieder ein

DataOutputStream
DateInputStream

Man kann zwar mit den normalen Streams problemlos byte-weise beliebige Daten schreiben und lesen, aber die Umwandlung der längeren Datentypen (`short`, `char`, `int`, `long`, `float`, `double`) obliegt dann den Programmierer:innen. Dies ist insofern problematisch, als z. B. die Reihenfolge der Bytes eines Long nicht plattformübergreifend festgelegt ist – Stichwort little und big endian.

Die Mühe der Umwandlung kann man sich sparen, wenn man `DataOutputStream` und `DataInputStream` verwendet. Hier sind für alle primitiven Datentypen und Strings entsprechende Methoden vorgesehen, um sie in einem einheitlichen Format zu lesen und zu schreiben.

3.17 Methoden von DataOutputStream [33]

- `void writeBoolean(boolean v)throws IOException`
- `void writeByte(int v)throws IOException`
- `void writeShort(int v)throws IOException`
- `void writeChar(int v)throws IOException`
- `void writeInt(int v)throws IOException`
- `void writeLong(long v)throws IOException`
- `void writeFloat(float v)throws IOException`
- `void writeDouble(double v)throws IOException`
- `void writeBytes(String s)throws IOException`
- `void writeChars(String s)throws IOException`
- `void writeUTF(String s)throws IOException`

3.18 Beispiel: DataOutputStream [35]

```
DataOutputStream out = new DataOutputStream(
       new BufferedOutputStream(
              new FileOutputStream("/tmp/daten")));

out.writeUTF("** Datendatei **");
out.writeUTF("Datum");
out.writeLong(new Date().getTime());
```

```
out.writeUTF("PI");
out.writeDouble(Math.PI);

out.close();
```

```
00 10 2A 2A 20 44 61 74 65 6E 64   ..** Datend
61 74 65 69 20 2A 2A 00 05 44 61   atei **..Da
74 75 6D 00 00 01 23 AF EE 3D 90   tum...#..=.
00 02 50 49 40 09 21 FB 54 44 2D   ..PI@.!.TD-
18                                 .
```

3.19 Methoden von DataInputStream [36]

- boolean readBoolean()throws IOException
- byte readByte()throws IOException
- int readUnsignedByte()throws IOException
- short readShort()throws IOException
- int readUnsignedShort()throws IOException
- char readChar()throws IOException
- int readInt()throws IOException
- long readLong()throws IOException
- float readFloat()throws IOException
- double readDouble()throws IOException
- String readUTF()throws IOException

3.20 Beispiel: DataInputStream [38]

```
DataInputStream dis = new DataInputStream(
       new BufferedInputStream(
              new FileInputStream("/tmp/daten")));

String header = dis.readUTF();
String datumTag = dis.readUTF();
Date datum = new Date(dis.readLong());
String PITag = dis.readUTF();
double pi = dis.readDouble();
dis.close();

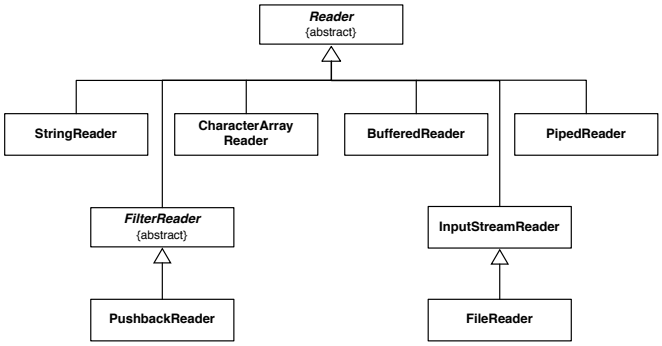
System.out.println(header);
System.out.println(datumTag + " " + datum);
System.out.println(PITag + " " + pi);
```

```
** Datendatei **
Datum Sun Sep 12 22:49:27 CEST 2010
PI 3.141592653589793
```

Kapitel 4

Reader / Writer

4.1 Klassenhierarchie: Reader [40]



Analog zu den Streams gibt es für das Lesen und Schreiben von Zeichenketten spezielle Klassen: **Reader** und **Writer**.

Reader
Writer

Ausgehend von einer abstrakten Basisklasse `Reader` gibt es für unterschiedlichen Speicherorte und Zwecke spezielle Subklassen. Diese sind:

- `StringReader` – aus Strings lesen — `StringReader`
- `CharacterArrayReader` – aus char-Arrays lesen — `CharacterArrayReader`
- `BufferedReader` – gepuffertes Lesen — `BufferedReader`
- `PipedReader` – Verbindung von Threads — `PipedReader`
- `InputStreamReader` – Verknüpfung von Stream und Reader — `InputStreamReader`
- `FileReader` – aus Dateien lesen — `FileReader`
- `FilterReader` – Filter — `FilterReader`
- `PushbackReader` – lesen und zurückstellen von Zeichen — `PushbackReader`

4.2 Methoden von Reader [42]

- `int read()` – liest ein Zeichen
- `int read(char[] cbuf)` – liest Zeichen in das `char`-Array

- `int read(char[] cbuf, int off, int len)` – liest maximal `len` Zeichen in das `char`-Array `cbuf` beginnend bei Position `off`
- `long skip(long n)` – überspringt `n` Zeichen
- `boolean ready()` – gibt an, ob der nächste `read()` auf dem Reader blockiert oder nicht
- `void mark(int readlimit)` – markiert die aktuelle Position im Reader und vergisst sie erst nach `readlimit` Bytes die daraufhin gelesen wurden
- `void reset()` – springt zu der mit `mark()` markierten Stelle im Reader zurück
- `boolean markSupported()` – zeigt an, ob der Reader überhaupt das setzen einer Marke mit `mark()` unterstützt
- `void close()` – schließt den Reader

Die Methoden des Reader entsprechen weitgehend denen der Streams, nur mit dem Unterschied, dass die Methoden `char` liefern und nehmen anstatt `byte`.

Auch hier ist wieder die Besonderheit zu beachten, dass die `read`-Methode den Rückgabetyp `int` hat, damit sie über den int-Wert -1 das Ende der Daten signalisieren kann. Deshalb darf man den Wert erst nach dem Vergleich mit -1 auf `char` casten.

4.3 Zeichen von einem Reader lesen [44]

```
Reader fr = new FileReader("/tmp/test.txt");

int daten;

while ((daten = fr.read()) > -1) {
    char c = (char) daten;
    // jetzt kann man etwas sinnvolles mit den Zeichen machen,
    // die aus der Datei gelesen wurden
}

fr.close();
```

```
Reader fr = new FileReader("/tmp/test.txt");

char b;

while ((b = (char) fr.read()) > -1) { // FEHLER!!
    // jetzt kann man etwas sinnvolles mit den Zeichen machen,
    // die aus der Datei gelesen wurden
}

fr.close();
```

Dieser Code funktioniert nicht. Die Methode `read()` signalisiert das Ende der Daten durch einen Rückgabewert von –1 als `int`. Wenn man das Ergebnis aber vorher auf `char` castet, wird das Lesen niemals beendet, da `char` ein vorzeichenloser Datentyp ist und daher das Ergebnis eines Casts auf `char` immer >= 0 ist.

4.4 Blöcke von einem Reader lesen [46]

```
Reader fr = new FileReader("/tmp/test.txt");

char[] daten = new char[1024];
int charactersRead;

while ((charactersRead = fr.read(daten)) > -1) {
    // jetzt kann man etwas sinnvolles mit den Zeichen machen,
    // die aus der Datei gelesen wurden
}

fr.close();
```

4.5 Beispiel: StringReader [47]

```
String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.\n" +
       "Nulla laoreet, sem vel mollis imperdiet, sapien mauris\n" +
       "sollicitudin arcu, sed viverra nulla dui at est.\n" +
       "Nunc est erat, semper id sollicitudin ut, pretium ac eros.\n";

Reader sr = new StringReader(text);
int gelesen;
while ((gelesen = sr.read()) > -1) {
    System.out.print((char) gelesen);
}

sr.close();
```

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Nulla laoreet, sem vel mollis imperdiet, sapien mauris
sollicitudin arcu, sed viverra nulla dui at est.
Nunc est erat, semper id sollicitudin ut, pretium ac eros.
```

StringReaders

Anstatt die Daten aus einer echten Quelle lesen, kann man mithilfe des **StringReaders** einen String als Quelle nehmen. Dies ist besonders angenehm für Tests, bei denen man auf so einfach die Funktionsweise z. B. eines Filter-Readers testen kann ohne Dateien anlegen zu müssen.

4.6 BufferedReader [48]

`BufferedReader` BufferedReader

- `BufferedReader(Reader in)`
- `BufferedReader(Reader in, int bufferSize)`
- `String readLine()`

Zu beachten ist, dass der `BufferedReader` die sehr nützliche `readLine()`-Methode einführt, mit der man Textdateien zeilenweise lesen kann. Das Ende der Datei wird bei dieser Methode durch den Rückgabewert `null` signalisiert.

4.7 Beispiel: BufferedReader [49]

```
String dateiname = "/tmp/text.txt";
BufferedReader reader = new BufferedReader(
       new FileReader(dateiname));

String zeile;
int nummer = 1;
while ((zeile = reader.readLine()) != null) {
    System.out.printf("%03d %s%n", nummer, zeile);
    nummer++;
}

reader.close();
```

```
001 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
002 Nulla laoreet, sem vel mollis imperdiet, sapien mauris
003 sollicitudin arcu, sed viverra nulla dui at est.
004 Nunc est erat, semper id sollicitudin ut, pretium ac eros.
```

4.8 Beispiel: Eigener FilterReader [50]

```
public class UpperCaseReader extends FilterReader {

    public UpperCaseReader(Reader in) {
        super(in);
    }

    @Override
    public int read(char[] cbuf, int off, int len) throws IOException {

        int result = super.read(cbuf, off, len);
```

```
        for (int i = off; i < off + result; i++) {
            cbuf[i] = Character.toUpperCase(cbuf[i]);
        }

        return result;
    }
}
```

```
String dateiname = "/tmp/text.txt";
BufferedReader reader = new BufferedReader(
        new UpperCaseReader(
                new FileReader(dateiname)));

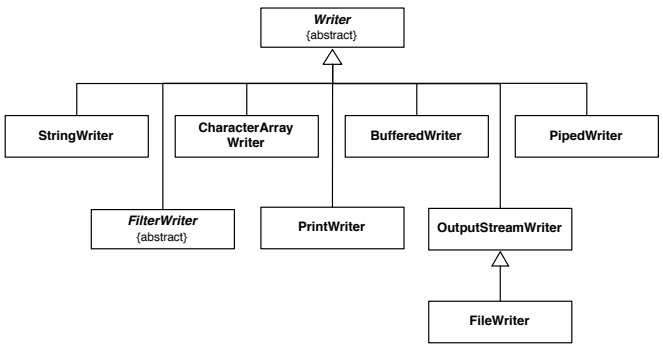
String zeile;

while ((zeile = reader.readLine()) != null) {
    System.out.println(zeile);
}

reader.close();
```

```
LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELIT.
NULLA LAOREET, SEM VEL MOLLIS IMPERDIET, SAPIEN MAURIS
SOLLICITUDIN ARCU, SED VIVERRA NULLA DUI AT EST.
NUNC EST ERAT, SEMPER ID SOLLICITUDIN UT, PRETIUM AC EROS.
```

4.9 Klassenhierarchie: Writer [52]



Analog zum `OutputStream` gibt es für das Schreiben von Zeichendaten die Subklassen von `Writer`. Diese sind:

- `StringWriter` – In Strings schreiben

`StringWriter`

- `CharacterArrayWriter` – In char-Arrays schreiben
- `BufferedWriter` – Gepuffertes Schreiben
- `PipedWriter` – Verbindung von Threads
- `OutputStreamWriter` – Verbindung von Stream und Writer
- `FileWriter` – In Datei schreiben
- `PrintWriter` – Formatierte Ausgabe von Datentypen
- `FilterWriter` – Filter

`CharacterArrayWriter`
`BufferedWriter`
`PipedWriter`
`OutputStreamWriter`
`FileWriter`
`PrintWriter`
`FilterWriter`

4.10 Die Methoden von Writer [54]

- `void write(int c)` – schreibt ein Zeichen
- `void write(char[] cbuf)` – schreibt alle Zeichen aus dem char-Array
- `void write(char[] cbuf, int off, int len)` – schreibt `len` Zeichen aus dem char-Array `cbuf` beginnend bei `off`
- `void write(String str)` – schreibt den String `str`
- `void write(String str, int off, int len)` – schreibt `len` Zeichen aus dem String `str`, beginnend bei `off`
- `Writer append(char c)` – schreibt ein Zeichen
- `void flush()` – leert interne Puffer und bringt die Daten auf die Platte
- `void close()` – schließt den Writer (impliziert einen `flush()`)

4.11 Zeichen in einen Writer schreiben [55]

```
Writer fw = new FileWriter("/tmp/mytext");

fw.write('T');
fw.write('e');
fw.write('x');
fw.write('t');
fw.write('\n');

fw.close();
```

```
> cat /tmp/mytext
Text
```

4.12 Strings in einen Writer schreiben [56]

```
Writer fw = new FileWriter("/tmp/mytext");

String daten = "Dies ist ein Text";

fw.write(daten);
fw.write(daten, 0, 5);
fw.write(daten, 0, 5);
fw.write(daten, 12, 5);
fw.write(daten, 12, 5);

fw.write("\n");

fw.close();
```



```
> cat /tmp/mytext
Dies ist ein TextDies Dies  Text Text
```



4.13 BufferedWriter [57]

BufferedWriter

- `BufferedWriter(Writer out)`
- `BufferedWriter(Writer out, int bufferSize)`
- `void newLine()`

4.14 Beispiel: CharArrayWriter [58]

```
CharArrayWriter cw = new CharArrayWriter();
PrintWriter pw = new PrintWriter(cw);

pw.printf("<html><head><title>%s</title></head>%n", "Toller Titel");
pw.printf("<body>%s</body></html>", "Ganz viel Body");

pw.close();

String webseite = cw.toString();

System.out.println(webseite);
```



```
<html><head><title>Toller Titel</title></head>
<body>Ganz viel Body</body></html>
```



4.15 InputStreamReader etc. [59]

- Manche APIs liefern nur Streams, manchmal möchte man die Daten aber mit einem Reader/Writer verarbeiten
  - Sockets
  - Servlet API
- `InputStreamReader` und `OutputStreamWriter` dienen dazu, Streams und Reader/Writer miteinander zu verbinden
- Verwendung als Filter bzw. Decorator

4.16 Beispiel: Einfacher HTTP-Client [60]

```
Socket socket = new Socket("www.web.de", 80);
OutputStream os = socket.getOutputStream();
InputStream is = socket.getInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(is));
Writer writer = new OutputStreamWriter(os);

writer.write("GET / HTTP/1.0\r\n\r\n");
writer.flush();

String line;
while ((line = reader.readLine()) != null) {
   System.out.println(line);
}

reader.close();
writer.close();
socket.close();
```

```
HTTP/1.1 200 OK
Date: Sun, 13 Sep 2010 08:47:59 GMT
Server: Apache/2
Last-Modified: Sun, 13 Sep 2010 08:45:25 GMT
ETag: "133970-18cc3-9202d340"
Accept-Ranges: bytes
Content-Length: 101571
Cache-Control: max-age=0
Expires: Sun, 13 Sep 2010 08:47:59 GMT
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="de" lang="de"
id="buster">
<head>
```

```
    <title>WEB.DE - E-Mail - Suche - DSL - Modem - Shopping
...
```

4.17 Die Klasse PrintWriter [62]

- Hat `print(...)` und `println(...)` Methoden für alle primitiven Datentypen, `char[]`, `String` und `Object`
- Die print-Methoden werfen keine `IOExceptions`, Fehler müssen mit `checkError` abgeholt werden
- Bietet mit `printf(...)` eine mächtige Methode zur formatierten Ausgabe von Daten

Kapitel 5

Konsolen IO

5.1 Konsole IO [64]

- Ein Sonderfall von Streams ist das Lesen und Schreiben von Standard-In und Standard-Out
- Normalerweise sind diese beiden Streams mit der **Konsole** (Bildschirm / Tastatur) verbunden
- Durch Ein- und Ausgabeumleitung können die Daten aber mit anderen Quellen und Senken verbunden sein
- Der Zugriff auf die Standard-Streams ist über `System` möglich
  - `System.out` - `PrintStream` der mit der Konsole verbunden ist (*stdout*)
  - `System.err` - `PrintStream` der mit der Konsole verbunden ist (*stderr*)
  - `System.in` - `InputStream` der mit der Konsole verbunden ist
- Man kann keine einzelne Zeichen mit `System.in.read()` lesen

Konsole

Für die Textausgabe besitzt jeder Computer eine sogenannte **Konsole**. Auf dieser können Daten ausgegeben und von ihr eingelesen werden. Für die Ausgabe gibt es zwei Kanäle **Standard-Out** (**stdout**) und **Standard-Error** (**stderr**). Wie die Namen schon deutlich machen, werden normale Ausgaben auf Standard-Out, Fehler aber auf Standard-Error geleitet. Obwohl beide Kanäle auf derselben Konsole herauskommen ist die Unterscheidung wichtig, wenn es zur **Ausgabeumleitung** kommt. Hier kann man nämlich die Kanäle getrennt in eine Datei oder ein anderes Programm umleiten: Die normalen Ausgaben werden umgeleitet, die Fehler bleiben aber sichtbar.

Konsole
Standard-Out
stdout
Standard-Error
stderr
Ausgabeumleitung

Für die Eingabe gibt es nur einen Kanal namens **Standard-In** (**stdin**), der normalerweise mit der Tastatur verbunden ist.

Standard-In
stdin

5.2 Beispiel: Echo [65]

```
public class Echo {
    public static void main(String[] args) throws IOException {
```

```
        BufferedReader br = new BufferedReader(
                new InputStreamReader(System.in));

        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    }
}
```

```
> cat /tmp/ipsum.txt | java -cp . pr22010.console.Echo
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Nulla laoreet, sem vel mollis imperdiet, sapien mauris
sollicitudin arcu, sed viverra nulla dui at est.
```

5.3 Beispiel: Umleitung von System.out [66]

```
PrintStream ps = new PrintStream("/tmp/umleitung.txt");
System.setOut(ps);

System.out.println("Hallo, das kommt ja gar nicht raus...");

ps.close();
```

```
> cat /tmp/umleitung.txt
Hallo, das kommt ja gar nicht raus...
```

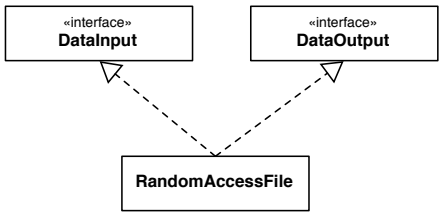
Kapitel 6

Random Access Files

6.1 Der Außenseiter RandomAccessFile [68]

- Streams bieten eine elegante Abstraktion, reichen manchmal aber nicht aus, um alle denkbaren Probleme zu lösen
- `RandomAccessFile` erlaubt den wahlfreien Zugriff auf Dateien, folgt aber nicht den Konventionen der Streams und ist nicht Teil der Klassenhierarchie der Streams
- Es hat ein anderes API, implementiert aber `DataInput` und `DataOutput`, genauso wie die `DataInputStream` und `DatatOutputStream`

`RandomAccessFile`



6.2 Methoden von RandomAccessFile [69]

- `int read()`, `int read(byte b[], int off, int len)`, `int read(byte b[])`
- `void readFully(byte b[])`, `void readFully(byte b[], int off, int len)`
- `int skipBytes(int n)`
- `void write(int b)`, `void write(byte b[])`, `void write(byte b[], int off, int len)`
- `long getFilePointer()`
- `void seek(long pos)`
- `long length()`
- `void setLength(long newLength)`
- `void close()`
- `boolean readBoolean()`, `int readInt()`, ...
- `String readLine()`

- `void writeBoolean(boolean v)`, `void writeInt(int i)`, ...

6.3 Die Modi von RandomAccessFile [71]

Beim Öffnen einer Datei mit RandomAccessFile muss man einen Modus mitgeben

- `"r"` – Nur lesender Zugriff
- `"rw"` – Lesender und schreibender Zugriff
- `"rws"` – wie „rw“ ohne Pufferung für den Inhalt und Metadaten
- `"rwd"` – wie „rw“ ohne Pufferung für den Inhalt

6.4 Beispiel: RandomAccessFile [72]

```
List<Student> studenten = new ArrayList<Student>();
studenten.add(new Student(100001L, 29));
...
RandomAccessFile db = new RandomAccessFile("/tmp/datenbank", "rw");

// Schreibe alle Matrikelnummern und Noten
for (Student student : studenten) {
    db.writeLong(student.matrikelNr);
    db.writeInt(student.note);
}

long length = db.length();
long datensatz = 0;
int besteNote = Integer.MAX_VALUE;
```

```
// Suche die beste Note
while (datensatz + 12 < length) {
    db.seek(datensatz + 8);
    int note = db.readInt();
    besteNote = Math.min(note, besteNote);
    datensatz += 12;
}
db.close();

System.out.println("Beste Note: " + (double) besteNote / 10);
```

```
00  00 00 00 00 00 01 86 A1 00 00 00 1D
0C  00 00 00 00 00 01 86 A2 00 00 00 1E
18  00 00 00 00 00 01 86 A3 00 00 00 1F
24  00 00 00 00 00 01 86 A4 00 00 00 18
30  00 00 00 00 00 01 86 A5 00 00 00 15
3C  00 00 00 00 00 01 86 A6 00 00 00 11
48  00 00 00 00 00 01 86 A7 00 00 00 0D
54  00 00 00 00 00 01 86 A8 00 00 00 0C
60  00 00 00 00 00 01 86 A9 00 00 00 0F
```

Kapitel 7

FileSystem und Path

7.1 Motivation für Abstraktion [75]

- Java ist plattformunabhängig, der Umgang mit Dateien ist aber extrem plattformabhängig
  - Unterschiedliche Separatoren (`/` vs. `\`)
  - Unterschiedliche Konzepte für Dateisystem-Wurzeln (`/` vs. `c:\`, `d:\`)
  - Unterschiedliche Berechtigungen (rwx vs. ACLs)
- Die Klassen **Path** und **FileSystem** abstrahieren von der Plattform und erlaubt es relativ abstrakt in Java mit Dateien zu operieren

Path
FileSystem

Es gibt neben den neuen Klassen `Path` und `FileSystem` noch eine alte Schnittstelle für den Umgang mit Dateien in Form der Klasse `java.io.File`. Diese ist inzwischen allerdings veraltet und wird nur noch dort verwendet, wo die Schnittstellen der Klassen explizit diese Klasse erwarten. Da man aber mit der Methode `toFile()` jedes `Path`-Objekt in ein `File`-Objekt umwandeln kann, gibt es keinen Grund mehr, die alten Klassen zu verwenden.

7.2 FileSystem und FileSystems [76]

Die Klasse `FileSystem` in `java.nio.file` stellt eine Abstraktionsschicht über dem Dateisystem dar und ermöglicht es, auf verschiedene Dateisysteme unabhängig von der zugrunde liegenden Implementierung zuzugreifen.

FileSystem

Die Funktionsweise der Klasse `FileSystem` besteht darin, dass sie eine Schnittstelle bereitstellt, um das Dateisystem zu durchsuchen, Dateien und Verzeichnisse zu erstellen und zu löschen, Metadaten abzurufen und andere Dateisystemoperationen durchzuführen. Die Klasse bietet auch die Möglichkeit, Dateisysteme zu konfigurieren und anzupassen, z. B. indem sie den Zugriff auf bestimmte Verzeichnisse einschränkt oder die Unterstützung für bestimmte Dateiformate aktiviert oder deaktiviert.

Ein Anwendungszweck der Klasse FileSystem ist es, es Entwickler:innen zu ermöglichen, plattformunabhängigen Code zu schreiben, der auf verschiedenen Betriebssystemen und Dateisystemen funktioniert.

`FileSystem`

- beschreibt ein Dateisystem
- abstrakte Klasse, die für die Dateisysteme implementiert wird
- `getPath()` liefert `Path`-Objekt für weitere Arbeit mit Dateisystem

`FileSystems`

Die Klasse `FileSystems` stellt Factory-Methoden zur Verfügung, um ein `FileSystem`-Objekt für ein bestimmtes Dateisystem zu erzeugen. Sie bietet auch eine Möglichkeit, auf spezielle Dateisysteme zuzugreifen, wie z. B. ein ZIP-Dateisystem. Es können `FileSystem`-Objekte für verschiedene Arten von Dateisystemen erstellt werden, einschließlich solcher, die in Dateien oder Netzwerkressourcen eingebettet sind.

`FileSystems`

- Factory-Klasse, um konkretes `FileSystem`-Objekt zu erzeugen
- `getDefault()` liefert `FileSystem` für aktuelles System
- `getFileSystem(URI uri)` liefert `FileSystem` für die gegebene URI

Dateisystem

Ein **Dateisystem** legt die Organisation und Speicherung von Daten auf einem Speichermedium wie einer Festplatte oder einem USB-Stick fest. Es definiert die Struktur und die Regeln, wie Daten gespeichert, abgerufen und geändert werden können. Das Dateisystem ermöglicht es, auf Dateien und Ordner zuzugreifen und diese zu manipulieren, und stellt sicher, dass Daten effizient und sicher auf dem Speichermedium gespeichert werden.

7.3 Path [77]

`Path`

Die Klasse `Path` in `java.nio.file` repräsentiert einen Pfad zu einer Datei oder einem Verzeichnis auf dem Dateisystem. Sie kann verwendet werden, um auf Dateien oder Verzeichnisse zuzugreifen und mit ihnen zu arbeiten, indem sie die erforderlichen Operationen wie Lesen, Schreiben und Löschen von Dateien bereitstellt.

Der Zweck der Klasse Path besteht darin, einen einheitlichen und plattformunabhängigen Weg zum Zugriff auf Dateien und Verzeichnisse bereitzustellen. Sie abstrahiert die Unterschiede in den Dateisystemen und bietet eine konsistente API, die unabhängig vom zugrunde liegenden Dateisystem funktioniert.

Die Klasse Path bietet auch eine Vielzahl von Methoden zur Manipulation von Pfaden, einschließlich der Konkatenation von Pfaden, der Umwandlung von Pfaden in andere Formate und der Ermittlung von Informationen über den Pfad, wie z. B. Dateiname, Verzeichnisname und Dateierweiterung.

`Path`

- repräsentiert den Pfad zu einer Datei / einem Verzeichnis
- bietet Methoden zum Arbeiten mit Dateien und Verzeichnissen

Beispiel mit Path

```
FileSystem fs = FileSystems.getDefault();
Path path = fs.getPath("/etc/passwd");
```

```
System.out.println(path.getFileName()); // -> passwd
System.out.println(path.getParent()); // -> /etc
System.out.println(path.getRoot()); // -> /

for (Path p : path) {
    System.out.print(p + ", "); // -> etc, passwd,
}
```

Wichtige Methoden von `Path`

- `static Path of(String first, String... more)`
  Kurzfassung für `FileSystems.getDefault(),getPath(...)`
- `static Path of(URI uri)`
  Neuer Pfad aus URI
- `Path resolve(...)`
  Pfad aus Komponenten zusammenbauen
- `boolean endsWith(...)`, `boolean startsWith(...)`
  Anfang und Ende des Pfades prüfen
- `boolean isAbsolute()`
  Testet of der Pfad absolut ist
- `Path getFileName()`
  Letzten Teil des Pfades bestimmen (den Dateinamen)
- `Path getParent()`
  Alles vor dem letzten Teil bestimmen
- `Path getRoot()`
  Wurzel des Dateisystems
- `FileSystem getFileSystem()`
  Referenz auf das dazugehörige Dateisystem
- `Path getName(int index)`
  Element an der Stelle `index`
- `int getNameCount()`
  Anzahl der Elemente
- `Path normalize()`
  Entfernt unnötige Teile aus dem Pfad
- `Path relativize(Path other)`
  Liefert einen neuen Pfad relativ zu `other`
- `Path toAbsolutePath()`
  Liefert einen absoluten Pfad
- `File toFile()`
  Konvertierung in `File`-Objekt

- `URI toUri()`
  Konvertierung in `URI`-Objekt

Beispiel für Path



```
// Pfad bezogen auf das Nutzer Home
Path path = Paths.get(System.getProperty("user.home"))
               .resolve("Pictures")
               .resolve("Urlaub2022");
System.out.println(path); // -> /home/thomas/Pictures/Urlaub2022

// Neuen, absoluten Pfad erzeugen
Path dwnl = path.getParent().resolve("..").resolve("Downloads");
System.out.println(dwnl); // -> /home/thomas/Pictures/../Downloads
System.out.println(dwnl.normalize()); // -> /home/thomas/Downloads
System.out.println(dwnl.isAbsolute()); // -> true

// Relativen Pfad erzeugen
Path videos = dwnl.relativize(Path.of("/home/thomas/Videos"));
System.out.println(videos); // -> ../Videos
System.out.println(videos.isAbsolute()); // -> false
```

7.4 Files [83]

Die Klassen für die Verarbeitung von Ein- und Ausgabe in Form von Streams (`InputStream`, `OutputStream`, `Reader`, `Writer`) sind objektorientiert elegant entworfen. Für viele Anwendungszwecke, z. B. um Daten komplett aus einer Textdatei zu lesen und dann zu verarbeiten, sind sie allerdings zu komplex und aufwändig zu benutzen. Aus diesem Grund bietet seit Java 1.7 die Klasse `Files` einen einfacheren Zugang zu den meisten Operationen mit Dateien. Sie stellt eine Sammlung von statischen Hilfs-Methoden zur Verfügung.

`Files`

- Datei-Operationen: Die Klasse bietet eine Reihe von Methoden zum Kopieren, Verschieben, Umbenennen und Löschen von Dateien und Verzeichnissen.
- Lesen und Schreiben von Dateien: Mit den Methoden der Klasse können Dateien gelesen und geschrieben werden, einschließlich der Möglichkeit, Zeilen aus einer Datei zu lesen und in eine Liste von Strings zu konvertieren.
- Verwaltung von Metadaten: Die Klasse kann auch verwendet werden, um Metadaten von Dateien und Verzeichnissen abzurufen und zu ändern, wie z. B. die Dateigröße, den letzten Zugriffszeitpunkt und die Berechtigungen.

`java.nio.file.Files` bietet einen Sammlung von Hilfsmethoden zum

- Lesen und Schreiben von Dateien
  `readAllBytes`, `readAllLines`, `readString`, `write`, `writeString`
- Kopieren von Daten
  `copy`

- Erzeugen von Dateisystemobjekten
  `createDirectory`, `createFile`, `createLink`, `createSymbolicLink`, `createTempDirectory`, `createTempFile`
- Verändern von Dateisystemobjekten
  `setLastModifiedTime`, `setOwner`, `setPosixFilePermissions`
- Testen und Lesen von Eigenschaften
  `isDirectory`, `isHidden`, `isExecutable`, `isReadable`, `isWritable`, `isRegularFile`, `size`
- Vergleichen von Dateien
  `mismatch`
- Verschieben von Dateien
  `move`
- Erzeugen von Streams
  `newInputStream`, `newOutputStream`
- …

Kapitel 8

Serialisierung

8.1 Das Problem [86]

- Wie transportiere ich Java Objekte über das Netzwerk?
- Wie schreibe ich Java Objekte in Dateien?
- Wie lese ich Java Objekte aus Dateien?
- Wie gehe ich mit ganzen Netzen von Objekten um?

In der Objektorientierten Programmierung entstehen schnell Netze von miteinander verbundenen Objekten. Wenn man diese über die Grenzen des Programms transportieren möchte, muss man das gesamte Netz durchlaufen, die Daten extrahieren und speichern. Beim Einlesen muss man aus den Daten die Objekte und ihre Beziehungen rekonstruieren. Dies kann – abhängig von der Größe des Objektnetzes – sehr aufwändig und kompliziert werden.

Es gibt generell zwei Gründe, warum man Objekte aus einem Java-Programm heraus holen möchte

- Man möchte die Daten an ein weiteres Java-Programm übergeben, dass zum selben Zeitpunkt läuft (**Marshalling**). Ein Beispiel wäre hier die Kommunikation über ein Netzwerk. **Marshalling**
- Man möchte die Daten speichern und später wieder in dasselbe Programm laden (**Persistenz**). **Persistenz**

In beiden Fällen muss man die komplexen Objektstrukturen im Speicher in eine einfache, zusammenhängende Folge von Bytes konvertieren. Da man hierbei die über Referenzen verbundenen, im Speicher nicht zusammen liegenden Daten hintereinander schreibt spricht man von **Serialisierung**. Die umgekehrte Operation, also den Aufbau der Objekte aus der linearen Folge von Bytes, nennt man **Deserialisierung**. **Serialisierung** **Deserialisierung**

8.2 Die Lösung: Java Serialisierung [87]

- Mit `DataOutputStream` und `DataInputStream` kann man Objekte in Streams schreiben und aus ihnen lesen, man muss aber die gesamte Logik von Hand schreiben
- Java-**Serialisierung** übernimmt dies automatisch und schreibt ganze Objektgeflechte **Serialisierung**

- Grundlegende Klassen
  - `ObjectOutputStream` - Schreiben von Objekten (*Serialisierung*)
  - `ObjectInputStream` - Lesen von Objekten (*Deserialisierung*)
- Serialisierung ist relativ robust gegen Änderungen an den Objekten, sodass serialisierte Objekte auch nach Änderung an der Klasse (häufig) noch deserialisiert werden können

Erfreulicherweise ist in Java bereits ein Mechanismus eingebaut, der es erlaubt, Objektnetze direkt in einen beliebigen Stream (Datei, Netzwerk etc.) zu schreiben. Die zentrale Klassen hierzu sind `ObjectOutputStream` und `ObjectInputStream`. Als Filter sind sie in der Lage, jeden anderen Stream als Quelle bzw. Senke der serialisierten Daten zu verwenden.

8.3 Regeln für die Serialisierung [88]

- Objekte können nur serialisiert werden, wenn ihre Klassen `java.io.Serializable` implementieren
- Superklassen müssen entweder selbst `java.io.Serializable` implementieren oder einen Default-Konstruktor haben
- Wenn die normale Serialisierung nicht ausreicht kann man mit `java.io.Externalizable` ein völlig eigenes Protokoll implementieren

Serialisierung kann Sicherheitslücken schaffen, wenn man in der Lage wäre jedes beliebige Objekt gegen seinen Willen zu serialisieren. So könnte man z. B. ein User-Objekt, das die Berechtigungen des Benutzers im Programm repräsentiert serialisieren, verändern und neu laden. Weiterhin gibt es Objekte, die sich aus logischen Gründen nicht serialisieren lassen: ein Thread ergibt später oder in einer anderen Java VM genauso wenig einen Sinn, wie eine geöffnete Datei oder eine Netzwerkverbindung.

`Serializable`
Tagging-Interface
zero abstract method interface

Der Einzige der entscheiden kann, ob Serialisierung erlaubt werden soll oder nicht ist die Klasse von der das zu serialisierende Objekt erzeugt wurde. Diese kann durch Implementierung des Interfaces `Serializable` anzeigen, dass ihre Objekte serialisiert werden dürfen. Das Interface hat selbst keine Methoden (**Tagging-Interface** oder **zero abstract method interface**) und dient nur dazu, die Klasse zu kennzeichnen.

Die Superklassen einer serialisierbaren Klasse müssen nicht zwingend selbst serialisierbar sein, sondern es reicht, wenn sie einen Default-Konstruktor haben. Eine andere Forderung wäre unlogisch, denn sonst müsste `Object` selbst serialisierbar sein und würde diese Eigenschaft (wegen der Transitivität der Implements-Beziehung) an *alle* Java-Klassen vererben, womit dann doch wieder jedes Java-Objekt serialisierbar wäre.

`Externalizable`

Wenn die Möglichkeiten der Serialisierung nicht ausreichen, kann man über das Interface `Externalizable` noch weiter in die Abläufe der Serialisierung eingreifen. Eine Behandlung würde aber hier den Rahmen sprengen.

8.4 Beispiel: Serialisierung [89]

```
Manager m = new Manager("Hans Alberts", 10000.0,
        new Date(0x2222222222L), "Buchhaltung");

ObjectOutputStream os = new ObjectOutputStream(
        new FileOutputStream("/tmp/albert.ser"));

os.writeObject(m);
os.close();
```

```
AC ED 00 05 73 72 00 1D 74 70 65 32 30 30 39 2E 73 65 72 69 61 6C 69 7A 61   ....sr..tpe2009.serializa
74 69 6F 6E 2E 4D 61 6E 61 67 65 72 30 B0 44 5C 0F C5 C3 3D 02 00 01 4C 00   tion.Manager0.D\...=...L.
09 61 62 74 65 69 6C 75 6E 67 74 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53   .abteilungt..Ljava/lang/S
74 72 69 6E 67 3B 78 72 00 21 74 70 65 32 30 30 39 2E 73 65 72 69 61 6C 69   tring;xr.!tpe2009.seriali
7A 61 74 69 6F 6E 2E 4D 69 74 61 72 62 65 69 74 65 72 93 04 FE B4 35 CB DC   zation.Mitarbeiter....5..
8F 02 00 03 44 00 06 67 65 68 61 6C 74 4C 00 07 67 65 62 6F 72 65 6E 74 00   ....D..gehaltL..geborent.
10 4C 6A 61 76 61 2F 75 74 69 6C 2F 44 61 74 65 3B 4C 00 04 6E 61 6D 65 71   .Ljava/util/Date;L..nameq
00 7E 00 01 78 70 40 C3 88 00 00 00 00 00 73 72 00 0E 6A 61 76 61 2E 75 74   .~..xp@.......sr..java.ut
69 6C 2E 44 61 74 65 68 6A 81 01 4B 59 74 19 03 00 00 78 70 77 08 00 00 00   il.Datehj..KYt....xpw....
22 22 22 22 22 78 74 00 0B 48 61 6E 73 20 41 6C 62 65 72 74 74 00 0B 42 75   """""xt..Hans Albertt..Bu
63 68 68 61 6C 74 75 6E 67                                                   chhaltung
```

Das Beispielprogramm legt ein Objekt vom Typ `Manager` an. Um dieses dann zu serialisieren, wird ein `ObjectOutputStream` angelegt. Da dieser zwar die Objekte serialisieren kann aber als Filter-Stream selbst nicht persistieren oder versenden, wird ein `FileOutputStream` für die eigentliche Speicherung verwendet.

`ObjectOutputStream`

Das Schreiben des Objektgeflechts erfolgt durch den Aufruf der Methode `writeObject()` des `ObjectOutputStream`. Hierdurch werden alle Objekte, die von der Referenz `m` aus erreichbar sind, gespeichert. Konkret in diesem Beispiel, der `Manager` selbst, zwei Strings und ein `Date`-Objekt.

8.5 Beispiel: Deserialisierung [90]

```
ObjectInputStream is = new ObjectInputStream(
        new FileInputStream("/tmp/albert.ser"));

Manager albert = (Manager) is.readObject();

is.close();

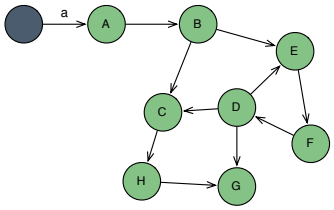
System.out.println(albert.getDetails());
```

```
Hans Alberts, geb. am Sat Aug 24 19:39:10 CET 1974 hat ein Gehalt von
10000.0 und leitet die Abteilung Buchhaltung
```

Die Deserialisierung erfolgt analog zur Serialisierung. Mithilfe eines `ObjectInputStreams` und eines darunterliegenden `FileInputStreams` werden die Objekte aus der Datei eingelesen und ein neues Objekt vom Typ `Manager` wird angelegt. Da die Methode `readObject()` den Rückgabetyp `Object` hat, muss die Referenz noch auf `Manager` gecastet werden. Die Ausgabe zeigt, dass tatsächlich wieder dieselben Daten im Objekt enthalten sind.

`ObjectInputStreams`

8.6 Objektgeflecht und Serialisierung [91]



Obwohl man der `writeObject()`-Methode nur eine einzige Referenz übergibt, werden bei der Serialisierung alle Objekte gespeichert, die von dieser Referenz aus erreichbar sind. Man sagt auch, dass die gesamte **transitive Hülle** (transitive closure) serialisiert wird. Dies ist nötig, da die Deserialisierung an einer ganz anderen Stelle zu einer ganz anderen Zeit passieren kann (z. B. bei einem gespeicherten Spielstand nach Wochen). Nur wenn man alle erreichbaren Objekte mit speichert, erhält man später ein korrektes Objekt-Modell.

transitive Hülle

Damit verhält sich Serialisierung anders als die `clone()`-Methode von `Object`. Während `clone()` nur eine *flache Kopie* (shallow copy) erzeugt, bekommt man durch eine Serialisierung und anschließende Deserialisierung eine **tiefe Kopie** (deep copy).

tiefe Kopie

8.7 Transiente Attribute [92]

- Serialisierung erfasst alle Felder, manche Objekte können aber nicht serialisiert werden
- Mit dem Schlüsselwort `transient` kann man einzelne Attribute von der Serialisierung ausnehmen (**transiente Attribute**)

transiente Attribute

Wie bereits oben erwähnt, gibt es einige Klassen, deren Objekte gespeichert oder übertragen sinnlos sind, z. B. Threads, geöffnete Dateien, Netzwerkverbindungen etc. Wenn diese in einem Objektgeflecht auftauchen, kann das ganze Gebilde nicht serialisiert werden. Es würde also eine einzige nicht serialisierbare Klasse ausreichen, damit das Speichern fehlschlägt. Um dieses Problem zu umgehen, gibt es die Möglichkeit, Attribute einer Klasse mit dem Schlüsselwort **transient** zu kennzeichnen. Diese Attribute werden bei der Serialisierung ignoriert und bei der Deserialisierung mit den entsprechenden Default-Werten (`null`, `0`, `false`) belegt. Außerhalb der Serialisierung verhalten sich transiente Attribute ganz normal.

transient

8.8 Beispiel: Transiente Felder [93]

```
public class A implements Serializable {
    private String s1;
    private String s2;
    private transient String s3;
    public A(String s1, String s2, String s3) {
        if ((s1 == null) || (s2 == null) || (s3 == null)) {
            throw new NullPointerException();
        }
        this.s1 = s1;
        this.s2 = s2;
        this.s3 = s3;
    }

    @Override
    public String toString() {
        return String.format("A [s1=%s, s2=%s, s3=%s]", s1, s2, s3);
    }
}
```



Im Beispiel ist das Attribut `s3` der Klasse `A` als `transient` gekennzeichnet. Weiterhin ist der Konstruktor so gestaltet, dass man keine Objekte anlegen kann, bei denen eines der Attribute `null` ist. Eine `toString()`-Methode gibt den Inhalt des Objektes aus.

```
// Object schreiben
ObjectOutputStream os = new ObjectOutputStream(
        new FileOutputStream("/tmp/transient.ser"));

os.writeObject(new A("String1", "String2", "String3"));
os.close();

// Object lesen
ObjectInputStream is = new ObjectInputStream(
        new FileInputStream("/tmp/transient.ser"));

A a = (A) is.readObject();
is.close();

System.out.println(a);
```



```
A [s1=String1, s2=String2, s3=null]
```



Wenn man ein Objekt vom Typ `A` serialisiert und danach wieder deserialisert, erhält man eine tiefe Kopie. Die Ausgabe zeigt, dass das Attribut `s3` tatsächlich nicht gespeichert wurde und

anstatt des Wertes „String3“ jetzt den Wert `null` hat. Das Schlüsselwort `transient` am Attribut `s3` hat also bewirkt, dass `s3` bei der Serialisierung ignoriert und bei der Deserialisierung auf `null` gesetzt wurde.

`transient`

Das Ergebnis ist insofern überraschend, als dass der Konstruktor von `A` es gar nicht zulässt, Objekte zu erzeugen bei denen eines der Attribute `null` ist. Die Ausgabe zeigt aber deutlich, dass genau dies für `s3` der Fall ist. Dieses seltsame Verhalten klärt sich, sobald man versteht, dass Serialisierung die Objekte nicht auf gewöhnlichem Wege erzeugt. Genauso wie das Clonen, geschieht die Erzeugung von Objekten bei der Deserialisierung am Konstruktor vorbei und passiert direkt im Speicher der VM. Somit sind Prüfungen durch den Konstruktor irrelevant, weil er überhaupt nicht aufgerufen wird.

8.9 Eigene Formate für die Serialisierung [95]

- Viele effiziente Speicherstrukturen (z. B. verkettete Listen) sind in der serialisierten Form nicht mehr kompakt
- Mit `readObject()` und `writeObject()` kann jede Klasse in die Serialisierung eingreifen

`readObject()`
`writeObject()`

Eine doppelt verkettete Liste ist eine sehr effiziente Struktur im Hauptspeicher, wenn man häufig Elemente einfügen und entfernen muss. Sie verwendet zwar mindestens drei Referenzen pro Element, dieser Speicherverbrauch wird aber in den meisten Fällen durch die Gewinne beim Verarbeiten der Listenelemente aufgewogen.

Serialisiert man eine verkettete Liste, so blähen sich die Daten erheblich auf. Neben den eigentlichen Nutzdaten wird die Struktur der Liste (einschließlich aller Referenzen) gespeichert, obwohl es genügen würde nur die Daten zu speichern, da in der serialisierten Form kein Bedarf am Einfügen oder Löschen von Elementen besteht und sowieso ein Element auf das nächste folgt.

Um Datenstrukturen effizient serialisieren zu können, bietet Java die privaten Methoden `writeObject()` und `readObject()` an. Diese werden – so vorhanden – bei der Serialisierung gerufen und erlauben es der Klasse anzugeben, in welchem Format sie am besten serialisiert werden sollte.

8.10 Beispiel: readObject() und writeObject() [96]

```
class Element implements Serializable {
    Element next;
    String daten;

    Element(String daten) {
        this.daten = daten;
    }
}
```

```
public class Liste implements Serializable {

    Element erstes;
    Element aktuelles;

    public Liste add(String daten) {
        if (erstes == null) {
            erstes = new Element(daten);
            aktuelles = erstes;
        } else {
            Element neu = new Element(daten);
            aktuelles.next = neu;
            aktuelles = neu;
        }
        return this;
    }
}
```

Das Beispiel zeigt eine sehr primitive, einfach verkettete Liste. Für jeden Eintrag in der Liste wird ein Objekt vom Typ `Element` verwendet. Die `Liste` kennt das erste und letzte Element und hat im vorliegenden Beispiel nur eine einzige Methode zum Anfügen von Elementen.

Wenn man diese Liste serialisiert, entstehen große Datenmengen, wie der folgende Auszug zeigt.

```
AC ED 00 05 73 72 00 26 74 70 65 32 30 30 39 2E 73 65 72 69 61 6C 69 7A 61   ....sr.&tpe2009.serializa
74 69 6F 6E 2E 72 65 61 64 6F 62 6A 65 63 74 2E 4C 69 73 74 65 43 D3 35 BF   tion.readobject.ListeC.5.
94 A2 85 C5 02 00 02 4C 00 09 61 6B 74 75 65 6C 6C 65 73 74 00 30 4C 74 70   .......L..aktuellest.0Ltp
65 32 30 30 39 2F 73 65 72 69 61 6C 69 7A 61 74 69 6F 6E 2F 72 65 61 64 6F   e2009/serialization/reado
62 6A 65 63 74 2F 4C 69 73 74 65 24 45 6C 65 6D 65 6E 74 3B 4C 00 06 65 72   bject/Liste$Element;L..er
73 74 65 73 71 00 7E 00 01 78 70 73 72 00 2E 74 70 65 32 30 30 39 2E 73 65   stesq.~..xpsr..tpe2009.se
72 69 61 6C 69 7A 61 74 69 6F 6E 2E 72 65 61 64 6F 62 6A 65 63 74 2E 4C 69   rialization.readobject.Li
73 74 65 24 45 6C 65 6D 65 6E 74 FA C4 5C A1 FD E9 11 C8 02 00 03 4C 00 05   ste$Element..\........L..
64 61 74 65 6E 74 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67   datent..Ljava/lang/String
3B 4C 00 04 6E 65 78 74 71 00 7E 00 01 4C 00 06 74 68 69 73 24 30 74 00 28   ;L..nextq.~..L..this$0t.(
4C 74 70 65 32 30 30 39 2F 73 65 72 69 61 6C 69 7A 61 74 69 6F 6E 2F 72 65   Ltpe2009/serialization/re
61 64 6F 62 6A 65 63 74 2F 4C 69 73 74 65 3B 78 70 74 00 01 44 70 71 00 7E   adobject/Liste;xpt..Dpq.~
00 02 73 71 00 7E 00 03 74 00 01 41 73 71 00 7E 00 03 74 00 01 42 73 71 00   ..sq.~..t..Asq.~..t..Bsq.
7E 00 03 74 00 01 43 71 00 7E 00 06 71 00 7E 00 02 71 00 7E 00 02 71 00 7E   ~..t..Cq.~..q.~..q.~..q.~
00 02                                                                        ..
```

Für jedes Elemente der Liste werden nicht nur die Referenzen, sondern auch noch der Datentyp als String gespeichert. Die Datenmenge explodiert.

Man kann diesem Problem aber begegnen, wenn man die Liste in die Serialisierung eingreifen lässt. Der folgende Quelltext zeigt wie.

```
public class Liste implements Serializable {
    transient Element erstes;
    transient Element aktuelles;
    ...

    private void writeObject(ObjectOutputStream os) throws IOException {
        os.defaultWriteObject();
```

```
        for (Element e = erstes; e != null; e = e.next) {
            os.writeUTF(e.daten);
        }
        os.writeUTF("");
    }
    ...
}
```

Über die Methode `writeObject()` erhält die Liste die Möglichkeit anzugeben, wie sie gespeichert werden möchte. Hierzu bekommt sie den `ObjectOutputStream` übergeben, der gerade für die Serialisierung verwendet wird. Anstatt die Elemente mit allen Referenzen zu schreiben, werden einfach die Elemente in der richtigen Reihenfolge gespeichert. Das Ende der Liste wird hier (ebenfalls wieder recht naiv) durch einen leeren String markiert – in einer echten Implementierung müsste man hier etwas intelligenter vorgehen.

Damit die für die Serialisierung nötigen Metadaten geschrieben werden, ist es sinnvoll am Anfang der `writeObject()`-Methode `os.defaultWriteObject()` aufzurufen.

Damit die vorhandenen Referenzen auf den Anfang und das Ende der Liste ignoriert werden, müssen diese `transient` gesetzt werden.

```
    private void readObject(ObjectInputStream is)
            throws IOException, ClassNotFoundException {

        is.defaultReadObject();
        String string;

        while ((string = is.readUTF()).length() > 0) {
            add(string);
        }
    }
```

```
AC ED 00 05 73 72 00 26 74 70 65 32 30 30 39 2E 73 65 72 69 61 6C 69 7A 61   ....sr.&tpe2009.serializa
74 69 6F 6E 2E 72 65 61 64 6F 62 6A 65 63 74 2E 4C 69 73 74 65 8E ED 12 B8   tion.readobject.Liste....
65 68 F9 AC 03 00 00 78 70 77 0E 00 01 41 00 01 42 00 01 43 00 01 44 00 00   eh.....xpw...A..B..C..D..
78                                                                            x
```

Analog zum Schreiben muss die Klasse über `readObject()` festlegen, wie die Daten gelesen werden sollen. Hier werden ebenfalls die Elemente der Liste linear eingelesen und über die `add()`-Methode wieder zu einer Liste im Speicher zusammengestellt.

Wie man deutlich sieht, hat sich die Datenmenge drastisch gegenüber der Standard-Serialisierung reduziert.

8.11 readResolve() [101]

- Serialisierung führt zu einer tiefen Kopie, da ganze Objektgeflechte kopiert werden
- Singletons sollen aber nicht mehrfach vorhanden sein
- Mit `readResolve()` kann eine Objekt sich selbst während der Deserialisierung ersetzen `readResolve()`
- Mit `writeReplace()` kann es während der Serialisierung eingreifen `writeReplace()`

Da die Serialisierung nicht weiß, welche Objekte Singletons sein sollen, werden diese genauso geschrieben und gelesen, wie alle anderen Objekte auch. Dies hat aber zur Folge, dass nach der Serialisierung und Deserialisierung eines Singletons, dieses plötzlich zweimal vorhanden ist.

Für diesen und ähnliche Fälle besteht die Möglichkeit für Klassen, über `writeReplace()` die eigene Instanz bei der Serialisierung zu ersetzen und bei `readResolve()` dasselbe bei der Deserialisierung zu machen.

8.12 Beispiel: readResolve() [102]

```
public class TrueSingleton implements Serializable {
    private static final TrueSingleton instanz = new TrueSingleton();

    private TrueSingleton() {
        // keine Instanzen zulassen
    }

    public static TrueSingleton getInstance() {
        return instanz;
    }
}
```

```
ObjectOutputStream os = new ObjectOutputStream(
        new FileOutputStream("/tmp/singleton.ser"));

os.writeObject(TrueSingleton.getInstance());
os.close();

ObjectInputStream is = new ObjectInputStream(
        new FileInputStream("/tmp/singleton.ser"));

TrueSingleton singleton = (TrueSingleton) is.readObject();
is.close();

System.out.println(singleton == TrueSingleton.getInstance()); // --> false
```

Das Beispiel zeigt, dass man von der Klasse `TrueSingleton` durch Serialisierung mehr als ein Objekt erzeugen kann. Dies widerspricht aber der Idee eines Singletons und sollte vermieden werden.

```
public class TrueSingleton implements Serializable {

    private static final TrueSingleton instanz = new TrueSingleton();

    public static TrueSingleton getInstance() {
        return instanz;
    }

    private Object readResolve() throws ObjectStreamException {
        return TrueSingleton.instanz;
    }
}
```

Durch die Methode `readResolve()` hat ein Objekt die Möglichkeit, sich selbst nach der Deserialisierung zu ersetzen. Anstatt des gelesenen Objektes wird das Objekt verwendet, dass von der Methode zurückgegeben wird.

8.13 serialVersionUID [105]

- Die Java-Laufzeit berechnet für jede Klasse eine spezielle ID um Kompatibilität bei der Serialisierung sicherzustellen, die **serialVersionUID** serialVersionUID
- Diese ID ist stark abhängig von diversen Faktoren
- Man sollte daher für serialisierbare Klassen eine eigene serialVersionUID angeben

```
public class Mitarbeiter implements Serializable {

    private static final long serialVersionUID = 8248054551788936332L;

    ...
}
```

Serialisierte Objekte können längere Zeit in dieser Form verharren, bevor sie wieder deserialisiert werden. Währenddessen kann es passieren, dass sich die Klasse verändert, zu der das Objekt gehört. Da die Klasse nicht serialisiert wird, können (persistentes) Objekt und Klasse sich unterscheiden und nicht mehr zusammenpassen. Um dies zu verhindern, erzeugt Java für jede Kasse eine Versionsnummer, die bei der Serialisierung zusammen mit dem Objekt abgespeichert wird. Diese Nummer verändert sich bei jeder Änderung an den Attributen der Klasse. Wird ein Objekt geladen, dessen Versionsnummer nicht zur Klasse passt, bricht die Deserialisierung ab.

Um den Mechanismus beeinflussen zu können, kann jede Java-Klasse ein privates, statisches Attribut namens `serialVersionUID` angeben, das dann anstatt der von Java berechnete Versionsnummer verwendet wird. Jetzt können die Programmierer:innen entscheiden, welche Änderungen an der Klasse dazu führen sollen, dass sich die Versionsnummer ändert. Solange man sie unverändert lässt, beschwert sich Java nicht und lädt die Objekte.

serialVersionUID

Es gilt gemeinhin als guter Stil, das Attribut zu setzen, um den Automatismus von Java auszuschalten. Andernfalls kann es passieren, dass gespeicherte Objekte nicht mehr geladen werden können, obwohl die Änderungen an der Klasse eigentlich unkritisch waren.

Kapitel 9

Ausblick: NIO

9.1 Die Probleme von java.io [107]

- Streams, Reader, Writer und RandomAccessFile sind nicht kopierfrei
- Alle IO Operationen sind **blockierend** (blocking) blockierend
- Moderne Konzepte wie Memory-Mapped-Files werden nicht unterstützt
- Mit Java 1.4 wurde parallel zu java.io mit java.nio ein neues IO-Konzept eingeführt (nio = new IO, gesprochen „neio“) mit folgenden Eingenschaften
  - kopierfrei
  - nicht blockierend
  - blockorientiert
  - memory-mapped-files werden unerstützt

Java NIO (New I/O) ist ein Paket in der Java-Plattform, das in der Version 1.4 eingeführt wurde und eine Alternative zum herkömmlichen I/O-System darstellt. Das Paket wurde entwickelt, um die Leistung von Eingabe- und Ausgabevorgängen in Java-Anwendungen zu verbessern und flexiblere und skalierbare Netzwerk- und Dateiverarbeitungsfunktionen bereitzustellen.

9.2 Channels und Buffer [108]

Die grundlegende Idee von Java NIO ist die Verwendung von sogenannten „non-blocking I/O“ Operationen, die eine asynchrone Verarbeitung ermöglichen. Im Gegensatz zum herkömmlichen I/O-System, können mit Java NIO mehrere I/O-Operationen parallel ausgeführt werden, ohne dass der Thread blockiert wird. Dies führt zu einer verbesserten Leistung und Skalierbarkeit von Java-Anwendungen.

Java NIO bietet auch eine flexiblere und leistungsstärkere Netzwerkverarbeitung durch die Verwendung von *Channels* und *Buffers*. Channels ermöglichen eine schnelle und direkte Verbindung zwischen der Anwendung und dem Netzwerk, während Buffers eine effiziente Möglichkeit bieten, Daten zu lesen und zu schreiben.

Ein weiterer Anwendungsbereich von Java NIO ist die Verarbeitung von Dateien. Mit dem Paket können Dateien schneller und effizienter gelesen und geschrieben werden, und es bietet

auch erweiterte Funktionen wie Dateisperren, um eine gleichzeitige Bearbeitung von Dateien durch mehrere Threads oder Prozesse zu verhindern.

Die zentralen Elemente von NIO sind

- `Buffer` – Verwaltet gelesene und zu schreibende Daten. Es gibt Buffer für alle primitiven Typen
- `ByteBuffer`, `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer`, `DoubleBuffer`
- `Channel` – Quelle bzw. Senke für Daten aus Buffers (ähnlich zu den Streams)
  - Können zum Lesen und Schreiben verwendet werden
  - Werden nicht direkt erzeugt, sondern über die Klassen aus der Standard IO-Library

`Channel`

In Java NIO (New I/O) ist ein `Channel` ein Objekt, das die Verbindung zwischen einer Quelle oder einem Ziel und der Anwendung darstellt. Es kann als eine Art Kanal oder Pipeline zwischen der Anwendung und der Quelle oder dem Ziel angesehen werden, über den Daten gelesen oder geschrieben werden können.

Die grundlegende Idee hinter Channels ist, dass sie eine schnelle und direkte Verbindung zwischen der Anwendung und der Quelle oder dem Ziel ermöglichen. Im Gegensatz zum herkömmlichen I/O-System, das eine langsame und schrittweise Verarbeitung von Daten durchführen kann, ermöglicht die Verwendung von Channels eine schnellere und effizientere Verarbeitung von Daten.

Channels bieten auch einige erweiterte Funktionen, wie beispielsweise Übertragungsoperationen, die eine schnelle Übertragung von Daten zwischen Kanälen ermöglichen. Dies kann insbesondere in Netzwerkverarbeitungsanwendungen nützlich sein, bei denen große Datenmengen schnell übertragen werden müssen.

9.3 Beispiel: Lesen in einen Buffer [109]

```
FileInputStream fis = new FileInputStream("/tmp/test.txt");

FileChannel fc = fis.getChannel();

ByteBuffer buffer = ByteBuffer.allocate(1024);

fc.read(buffer);

fis.close();
```

9.4 Beispiel: Schreiben aus einem Buffer [110]

```
FileOutputStream fos = new FileOutputStream("/tmp/output.txt");
FileChannel fc = fos.getChannel();
```

```
ByteBuffer buffer = ByteBuffer.allocate(128);

buffer.put("Hello World\n".getBytes());

buffer.flip();

fc.write(buffer);

fos.close();
```

```
>cat /tmp/output.txt
Hello World
```



9.5 Buffer [111]

- Buffer werden über statische Factory-Methoden angelegt, z. B. `ByteBuffer buffer = ByteBuffer.allocate(1024);`
- Jeder Buffer hat drei wichtige Eigenschaften
  - `capacity` – Größe des Buffers, d. h. die Anzahl der Bytes etc. die er fassen kann
  - `position` – Aktuelle Position des Lese- oder Schreibzeigers im Buffer
  - `limit` – Grenze bis zu der der Buffer ausgelesen werden kann
- Zusätzlich kann man, wie bei Streams, eine Marke (`mark`) setzen, so der Buffer dies unterstützt
- Es gilt die Invariante `mark <= position <= limit <= capacity`
- `DirectBuffer` greifen direkt auf Betriebssystemspeicher zu

Buffer

Ein `Buffer` ist ein Objekt, das verwendet wird, um Daten temporär zu speichern und die Übertragung von Daten zwischen Channels und Anwendungen zu erleichtern. Ein Buffer kann als eine Art Puffer oder Zwischenspeicher betrachtet werden, der zwischen dem Channel und der Anwendung platziert wird, um Daten zu lesen oder zu schreiben.

Ein Buffer wird erstellt, um eine bestimmte Menge an Daten aufzunehmen, und kann dann verwendet werden, um Daten aus dem Channel zu lesen oder in den Channel zu schreiben. Der Buffer speichert die Daten temporär, bevor sie in den Channel geschrieben oder aus dem Channel gelesen werden.

9.6 Beispiel: Kopierprogramm [112]

```
// Quell und Zieldatei
String quelle = "/tmp/test.txt";
String ziel = "/tmp/ziel.txt";
```



```
// Streams und Channels für Quell und Zieldatei
FileInputStream in = new FileInputStream(quelle);
FileOutputStream out = new FileOutputStream(ziel);
FileChannel fcin = in.getChannel();
FileChannel fcout = out.getChannel();

ByteBuffer buffer = ByteBuffer.allocate(1024); // Puffer für Dateiinhalt
```

```
// Daten aus Quell- in Zieldatei kopieren
while (fcin.read(buffer) >= 0) {
    buffer.flip();
    fcout.write(buffer);
    buffer.clear();
}
// Streams schließen (channels werden automatisch geschlossen)
in.close();
out.close();
```



9.7 Methoden von Buffer [114]

- Lesen
  - `byte get()`
  - `ByteBuffer get(byte dst[])`
  - `ByteBuffer get(byte dst[], int offset, int length )`
  - `byte get(int index)`
- Schreiben
  - `ByteBuffer put(byte b)`
  - `ByteBuffer put(byte src[])`
  - `ByteBuffer put(byte src[], int offset, int length)`
  - `ByteBuffer put(ByteBuffer src)`
  - `ByteBuffer put(int index, byte b)`

9.8 Weitere Features von NIO [116]

- Memory mapped files
  `MappedByteBuffer mbb = fc.map(MapMode.READ_WRITE, 0, 1024);`
- File Locks
  `FileLock lock = fc.lock(start, end, false);`
- Asynchrone Socket I/O
- Automatisches Aufteilen oder Zusammenführen von Buffern
  - `ScatteringByteChannel`

- GatheringByteChannel

Index

Ausgabeumleitung, 24

blockierend, 45
BufferedInputStream, 12
BufferedOutputStream, 12

DataOutputStream, 12
DateInputStream, 12
Dateisystem, 30
Decorator Pattern, 11
Deserialisierung, 34

Eingabe/Ausgabe, 2

FileSystem, 29
Filter-Streams, 11

I/O, 2
InputStreams, 3

Konsole, 24

Marshalling, 34

OutputStreams, 3

Path, 29
Persistenz, 34
puffern, 12

Reader, 3, 15

Serialisierung, 34
serialVersionUID, 43
Standard-Error, 24
Standard-In, 24
Standard-Out, 24
stderr, 24
stdin, 24
stdout, 24
Streams, 3
StringReaders, 17

Tagging-Interface, 35
tiefe Kopie, 37
transient, 37
transiente Attribute, 37
transitive Hülle, 37

Writer, 3, 15

zero abstract method interface, 35

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Generische Typen



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**
- 1.1 Fragen [5] . . . 1
- 1.2 Begriffe [6] . . . 1
- 1.3 Links zur Java-Insel [8] . . . 2

2 Motivation und Einführung **3**
- 2.1 Motivation für Generics [10] . . . 3
- 2.2 Beispiel: Einfacher Stack [11] . . . 3
- 2.3 Beispiel: Einfacher Stack für Strings [13] . . . 5
- 2.4 Beispiel: Einfacher Stack für Integer [14] . . . 5
- 2.5 Eigenschaften von Generics [15] . . . 6
- 2.6 Beispiel: Stack mit Generics [16] . . . 7
- 2.7 Darstellung in UML [18] . . . 8
- 2.8 Verbesserte Typ-Inferenz [19] . . . 8

3 Generische Typen in Java **11**
- 3.1 Begriffe [22] . . . 11
- 3.2 Generics und Vererbung [23] . . . 11
- 3.3 Beispiel: Generische Vererbung [24] . . . 12
- 3.4 Beispiel: Vererbung mit gesetztem Parameter [26] . . . 12
- 3.5 Generics als Compile-Zeit-Konstrukt [28] . . . 13
- 3.6 Beispiel: Type Erasure [29] . . . 14
- 3.7 Invarianz [30] . . . 14
- 3.8 Beispiel: Invarianz [31] . . . 15
- 3.9 Generics vs. Arrays [32] . . . 15
- 3.10 Beispiel: Kovarianz vs. Invarianz [33] . . . 17
- 3.11 Beispiel: Reified vs. Erasure (Array) [34] . . . 17
- 3.12 Beispiel: Reified vs. Erasure (Generic) [35] . . . 17

4 Wildcards und gebundene Parameter **19**
- 4.1 Wildcards [37] . . . 19
- 4.2 Beispiel: Wildcard [38] . . . 20
- 4.3 Beispiel: Bound Wildcard [39] . . . 21
- 4.4 Wildcard mit extends [42] . . . 22
- 4.5 Beispiel: Wildcard mit extends [43] . . . 22

4.6 Wildcard mit super [44] . . . 23
4.7 Beispiel: Wildcard mit super [45] . . . 23
4.8 Beispiel: super und extends [46] . . . 24
4.9 Gebundene Parameter [48] . . . 24
4.10 Beispiel: Gebundene Parameter [49] . . . 25

5 Weitere Themen 26
5.1 Beispiel: Multiple Typ-Parameter [52] . . . 26
5.2 Generische Methoden [53] . . . 27
5.3 Beispiel: Generische Methoden [54] . . . 27
5.4 Beispiel: Generische Methoden (ohne Typ-Inferenz) [56] . . . 28

Index ii

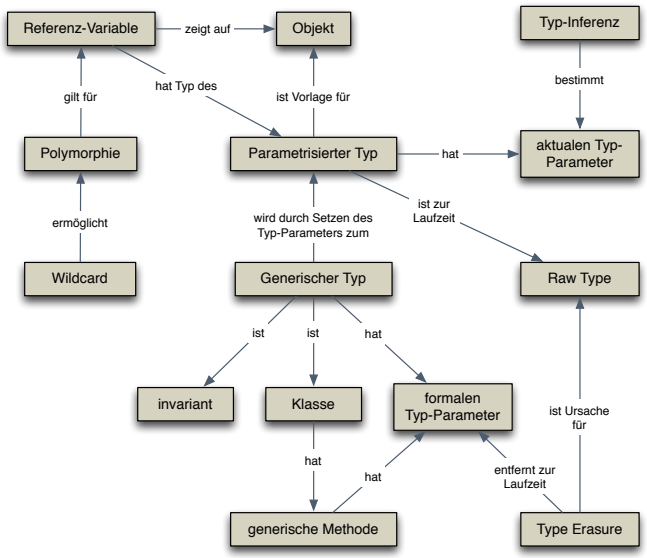
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie kann ich Klassen schreiben, wenn ich den Typ der verarbeiteten Daten noch gar nicht kenne?
- Wie kann ich mit einer Klasse verschiedene Datentypen verarbeiten?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

Name	Beispiel
Generischer Typ (generic type)	`List<E>`
Formaler Typ-Parameter (formal type parameter)	`E`
Parametrisierter Typ (parametrized type)	`List<String>`
Aktualer Typ-Parameter (actual type parameter)	`String`
Ungebundener Wildcard-Typ (unbounded wildcard type)	`List<?>`
Gebundener Wildcard-Typ (bounded wildcard type)	`List<? extends Number>`
Raw Type	`List`
Gebundener Typ-Parameter (bounded type parameter)	`<E extends Number>`
Rekursiv gebundener Typ-Parameter (recursive bounded type)	`<T extends Comparable<T>>`
Generische Methode (generic method)	`static <E> List<E> asList(E[] a)`

1.3 Links zur Java-Insel [8]

- 12. Generics<T>

Kapitel 2

Motivation und Einführung

2.1 Motivation für Generics [10]

Wie abstrahiere ich vom konkreten Typ eines Objekts?

- Wie kann ich (typsichere) **Containerklassen** bauen, die jedes beliebige Objekt aufnehmen können? **Containerklassen**
- Wie baue ich typsichere Frameworks?
- Wie vermeide ich Casts und mache dadurch meinen Code typsicher?

In der fortgeschrittenen Programmierung kommt es häufig vor, dass man Klassen schreibt, die dafür zuständig sind Objekte zu verwalten (**Containerklassen**). So fallen z. B. die Datenstrukturen Stack, Liste, Tabelle in diese Kategorie. Jetzt ist es bei der Entwicklung einer solchen Klasse eigentlich egal, ob letztendlich Strings oder Datumsobjekte verwaltet werden sollen. Die grundlegenden Operationen eines Stacks ändern sich nicht. **Containerklassen**

Mit den bisher bekannten Programmiertechniken kann man solche Datenstrukturen zwar bauen, wird aber letztendlich immer auf `Object` als kleinsten gemeinsamen Nenner zurückgreifen. D. h. in allen Schnittstellen dieser Klasse taucht `Object` als Parameter bzw. Rückgabetyp auf.

Die Verwendung von `Object` ist aber unbefriedigend, da

1. die Klassen dann nicht typsicher sind (man kann jedes beliebige Objekt auf den Stack legen),
2. man beim Herausholen der Objekte ständig casten muss.

Gesucht ist also ein Mechanismus, der es erlaubt diese Probleme zu umgehen.

2.2 Beispiel: Einfacher Stack [11]

```
public class SimpleStack {

    private Object[] stack;
    private int pos;

    public SimpleStack(int size) {
```

```
        stack = new Object[size];
        pos = 0;
    }

    public void push(Object o) {
        stack[pos++] = o;
    }

    public Object pop() {
       return stack[--pos];
    }
}
```

Der hier dargestellte Stack ist ausgesprochen schlecht programmiert. Er überprüft Über- und Unterschreitungen der Array-Grenzen nicht und es fehlen viele wichtige Methoden, wie z. B. `swap()`, `peek()`, `clear()` etc., die eine gute Stack-Klasse haben sollte. Für die Zwecke dieses Kapitels ist er aber ausreichend und besser verständlich als eine vollständige und korrekte Implementierung.

Wir werden später noch sehen, dass es in der Java-Klassenbibliothek bereits eine sehr gute Implementierung eines Stacks gibt (`java.util.Deque`), sodass es ohnehin meistens keinen Sinn ergibt, eigene Stack-Implementierungen zu programmieren.

Die Klasse `SimpleStack` ist jetzt eine Containerklasse, wie sie weiter oben beschrieben wurden: sie kann andere Objekte verwalten und verwendet `Object`, damit man sie für beliebige Arten von Objekten einsetzen kann.

```
SimpleStack stack = new SimpleStack(10);

stack.push("Hello");
String s = (String) stack.pop();
System.out.println(s);

stack.push(42);
s = (String) stack.pop(); // FEHLER!!
System.out.println(s);
```

```
Hello
Exception in thread "main" java.lang.ClassCastException: ↩
    java.lang.Integer
     at pr2.Verwender.main(Verwender.java:13)
```

Die fehlende Typsicherheit von `SimpleStack` hat zur Konsequenz, dass die Klasse beim Einfügen von Objekten nicht prüfen kann, ob diese den richtigen Typ haben. Dies fällt erst beim Herausholen auf, wenn der entsprechende Cast fehlschlägt und führt zu einem *Laufzeitfehler*.

Das Risiko eines Laufzeitfehlers kann man umgehen, wenn man für jeden Datentyp, der verarbeitet werden soll, eine eigene Klasse schreibt, also einen Stack für Strings, einen für Integer, Datum etc.

2.3 Beispiel: Einfacher Stack für Strings [13]

```
public class SimpleStackString {

    private String[] stack;
    private int pos;

    public SimpleStackString(int size) {
        stack = new String[size];
        pos = 0;
    }

    public void push(String o) {
        stack[pos++] = o;
    }

    public String pop() {
       return stack[--pos];
    }
}
```

Durch Änderung der Datentypen in den Methoden auf `String` kann man die Klasse `SimpleStack` so umwandeln, dass sie typsicher nur noch Strings verwaltet und so die oben beschriebene Fehlerquelle beim Cast wegfällt.

2.4 Beispiel: Einfacher Stack für Integer [14]

```
public class SimpleStackInteger {

    private Integer[] stack;
    private int pos;

    public SimpleStackInteger(int size) {
        stack = new Integer[size];
        pos = 0;
    }

    public void push(Integer o) {
        stack[pos++] = o;
    }

    public Integer pop() {
```

```
        return stack[--pos];
    }
}
```

Genauso kann man die Klasse für Integer-Werte anpassen.

Vergleicht man `SimpleStack`, `SimpleStackString` und `SimpleStackInteger` so wird sofort deutlich, dass sie sich nur an drei Stellen unterscheiden:

1. dem Typ des Arrays `stack`
2. dem Typ des Parameters der `push`-Methode
3. dem Typ des Rückgabewertes der `pop`-Methode

In allen drei Fällen besteht der Unterschied also nur in dem Typ einer Variable bzw. dem Rückgabetyp einer Methode.

Elegant wäre es, wenn man hier den Typ konfigurierbar machen könnte, d. h. wenn man erst bei der Verwendung eines `SimpleStack` sagen könnte, welchen Typ die drei Elemente haben sollten. Genau dies leisten Generics.

2.5 Eigenschaften von Generics [15]

Generics erlauben vom konkreten Typ zu abstrahieren

Generics

- Der Typ einer Variable, eines Parameters, eines Rückgabewertes etc. ist selbst ein Variable **Typ-Variable** (type variable)

  Typ-Variable
- Klassen und Methoden haben zusätzliche **Typ-Parameter** (type parameter), welche die Typ-Variablen setzen

  Typ-Parameter
- Der Verwender kann Typ-Parameter setzen und damit die Klassen/Methoden parametrieren
- Klassen mit Typ-Parametern sind **generische Typen** (generic types) bzw. generische Klassen (generic classes)

  generische Typen
- Methoden mit Typ-Parametern sind **generische Methoden** (generic methods)

  generische Methoden

Durch die Verwendung von Generics kann man Typen konfigurierbar machen. D. h. welchen Typ eine Variable, oder ein Parameter hat, wird erst später festgelegt – der Typ selbst wird variable. Damit dies funktioniert, muss man eine Variable haben, die festlegt, welchen Typ ein Parameter haben soll, man spricht hier von einer *Typ-Variable*.

Wichtig ist jetzt der Unterschied zwischen Variablen und Typ-Variablen, da sie auf zwei unterschiedlichen Abstraktionsebenen liegen:

- Eine *Typ-Variable* legt fest welchen Typ ein Element, z. B. ein Parameter haben soll. Sie kann nur einen *Typ* (z. B. `String`, `Integer`) enthalten.
- Eine *Variable* hat einen Typ (der möglicherweise durch eine Typ-Variable festgelegt wurde) und enthält dann einen *Wert* (z. B. `"Hallo"` oder `65`), der zu dem Typ, den sie hat, passen muss.

Zuerst legt also die Typ-Variable den Typ einer Variable fest. Die Variable trägt dann einen Wert entsprechend ihres gerade festgelegten Typs.

Die Typ-Variable muss gesetzt werden, hierzu dient der *Typ-Parameter*. So wie ein Parameter den Wert einer Variable bestimmt, bestimmt der Typ-Parameter den Wert einer Typ-Variable.

Wenn jetzt eine Klasse einen Typ-Parameter besitzt, spricht man von einer *generischen Klasse* oder einem *generischen Typ*. Bei Methoden analog von einer *generischen Methode*.

2.6 Beispiel: Stack mit Generics [16]

```
public class SimpleStack<T> {
    private T[] stack;
    private int pos;

    @SuppressWarnings("unchecked")
    public SimpleStack(int size) {
        stack = (T[]) new Object[size];
        pos = 0;
    }

    public void push(T o) {
        stack[pos++] = o;
    }

    public T pop() {
       return stack[--pos];
    }
}
```

Das Beispiel zeigt den `SimpleStack` als generischen Typ. Sofort fällt das <T> nach dem Klassennamen auf. Hierbei handelt es sich um die Deklaration eines Typ-Parameters `T` für die Klasse `SimpleStack`. Der Name des Parameters kann frei gewählt werden, nur einen Buchstaben zu verwenden hat sich aber etabliert. Analog zu den Parametern einer Methode handelt es sich hier um den **Formalparameter**. Da wir bei den Typ-Parametern schon auf der höchsten Abstraktionsebene angekommen sind, hat der Typ-Parameter selbst keinen Typ, d. h. anders als bei einem Methodenparameter, der immer mit Typ deklariert wird (z. B. `add(int a, int b)`), wird dem Typ-Parameter kein Typ vorangestellt.

Formalparameter

Innerhalb der Klasse wird an allen Stellen, an denen der Typ abhängig vom Typ-Parameter festgelegt werden soll `T` verwendet. Hierbei handelt es sich nicht um die Deklaration des Parameters, sondern seine Verwendung. Genauso, wie in einer Methode `add(int a, int b)` zwei Parameter deklariert und dann innerhalb der Methode verwendet werden (`return a + b`).

Innerhalb des Konstruktors gibt es eine widersinnige Konstruktion (`stack = (T[])new Object[size];`) bei der ein `Object`-Array angelegt wird und dann auf den Typ `T` gecastet. Den Hintergrund können wir an dieser Stelle noch nicht liefern. Er wird aber klar, wenn über den

Zusammenhang von Generics und Arrays gesprochen wird und die Type-Erasure eingeführt wurde. Daher müssen wir diese Konstruktion vorläufig einfach so hinnehmen.

```
var stack = new SimpleStack<String>(10);

stack.push("Hello");
String s = stack.pop();
System.out.println(s);

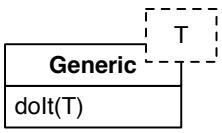
stack.push(42); // FEHLER!!
s = stack.pop();
System.out.println(s);
```

```
pr2/Verwender.java:12: push(java.lang.String) in pr2.SimpleStack
     <java.lang.String> cannot be applied to (int)
        stack.push(42);
             ^
```

Der Verwender unseres neuen Stacks gibt beim Anlegen der Referenz `stack` und bei der Erzeugung des Objektes an, welchen Wert der Typ-Parameter haben soll, es wird also der Aktual-Parameter gesetzt (hier `String`).

Versucht man jetzt ein Objekt auf den Stack zu legen, das nicht dem gesetzten Typ (hier `String`) entspricht, kommt es zu einer Fehlermeldung durch den Compiler. Der Laufzeitfehler aus dem ursprünglichen Beispiel ist also verschwunden und wir merken bereits beim Programmieren, dass 42 nicht auf dem Stack gespeichert werden kann.

2.7 Darstellung in UML [18]



In UML wird ein generischer Typ durch eine zusätzliche Angabe des Typ-Parameters gekennzeichnet.

2.8 Verbesserte Typ-Inferenz [19]

Seit Java 7 ist die Typ-Inferenz für Generics deutlich verbessert worden

- In den meisten Situationen muss man den Typ-Parameter nur noch bei der Deklaration setzen
- Bei der Objekterzeugung kann die Angabe des Typ-Parameters durch den **diamond** <> diamond

ersetzt werden

```
List<String> list = new ArrayList<>();
Map<Integer, String> map = new HashMap<>();
Set<?> set = new HashSet<>();
```

Seit Java 7 braucht der Typ-Parameter nicht angegeben zu werden, wenn er sich aus dem Kontext ergibt. Da es keine Polymorphie im klassischen Sinne bei generischen Typen gibt, kann in den allermeisten Fällen der Typ-Parameter bei der Objekterzeugung aus dem Typ-Parameter der Referenzvariable abgeleitet werden. Vor Java 7 musste man den Parameter sowohl bei der Deklaration der Referenzvariablen als auch bei der Objekterzeugung angeben.

Java 6

```
SimpleStack<String> stack = new SimpleStack<String>(10);
```

Java 7

```
SimpleStack<String> stack = new SimpleStack<>(10);
```

Wenn sich der Typ-Parameter nicht eindeutig ergibt, muss er allerdings auch bei Java 7 im Zusammenhang mit der Objekterzeugung noch einmal festgelegt werden.

Seit Java 10 kann man das neue Schlüsselwort `var` verwenden, das den Compiler anweist, den Datentyp der Variablen aus dem Datentyp des Wertes abzuleiten, mit dem die Variable initialisiert wird. In diesem Fall muss man den Typ-Parameter aber wieder angeben.

Generische Typen und `var` `var`

- entweder man gibt den Typ rechts wieder an *oder*
- der Typ-Parameter wird automatisch auf `Object` gesetzt (meist unerwünscht)

Java 10

```
var stack = new SimpleStack<String>(10);
```

Lässt man den Typ-Parameter auch auf der rechten Seite weg, gibt es keinen Compile-Fehler, sondern der Typ wird auf `Object` gesetzt.

Fehlender Typ

```
var stack = new SimpleStack<>(10);
```

Es ist allerdings zu beachten, dass man bei Verwendung von `var` nicht mehr mit Polymorphie arbeiten kann, da der Typ der Variablen vom Compiler anhand der Zuweisung ermittelt wird.

Im Zusammenhang mit Collections (siehe weiter unten) ist aber gerade die Verwendung des Interfaces als Typ der Variable sinnvoll, da man so flexibler bleibt.

```
// la ist vom statischem Typ List
List<String> la = new ArrayList<>();

// lb ist vom statischem Typ ArrayList
var lb = new ArrayList<String>();
```



Kapitel 3

Generische Typen in Java

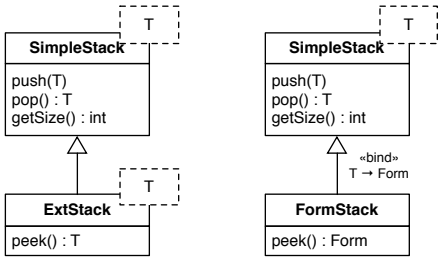
3.1 Begriffe [22]

Name	Beispiel
Generischer Typ (generic type)	`List<E>`
Formaler Typ-Parameter (formal type parameter)	`E`
Parametrisierter Typ (parametrized type)	`List<String>`
Aktualer Typ-Parameter (actual type parameter)	`String`
Ungebundener Wildcard-Typ (unbounded wildcard type)	`List<?>`
Gebundener Wildcard-Typ (bounded wildcard type)	`List<? extends Number>`
Raw Type	`List`
Gebundener Typ-Parameter (bounded type parameter)	`<E extends Number>`
Rekursiv gebundener Typ-Parameter (recursive bounded type)	`<T extends Comparable<T>>`
Generische Methode (generic method)	`static <E> List<E> asList(E[] a)`

Die Tabelle zeigt eine ganze Reihe von Begriffen, die im Zusammenhang mit generischen Typen relevant sind. Die meisten sind bis jetzt noch unbekannt und sollen in den folgenden Abschnitten beschrieben werden.

3.2 Generics und Vererbung [23]

- Generische Klassen können Subklassen haben
- Entweder
  - ist die Subklasse selbst wieder generisch oder
  - die Subklasse setzt den Typ-Parameter und ist damit nicht generisch



Wenn man von einer generischen Klasse erbt, hat man zwei Möglichkeiten: (a) Der Subtyp ist selbst ein generischer Typ oder (b) man setzt beim Erben den Typ-Parameter, sodass die Subklasse nicht mehr generisch ist.

3.3 Beispiel: Generische Vererbung [24]

```
public class ExtStack<T> extends SimpleStack<T> {

    public ExtStack(int size) {
        super(size);
    }

    public T peek() {
        T wert = pop();
        push(wert);
        return wert;
    }
}
```

In diesem Beispiel ist die Subklasse selbst wieder generisch und gibt den Typ-Parameter weiter an die Basisklasse. Für das Verständnis wichtig ist die Tatsache, dass hier zwei verschiedene <T> zu sehen sind:

1. Das erste `T` (`ExtStack<T>`) ist der Typ-Parameter, der dann durch den Verwender gesetzt wird, z. B. durch `ExtStack<String> s = new ExtStack<String>(10)`.
2. Das zweite `T` ist bereits die erste Verwendung des Typ-Parameters, der an die Klasse `SimpleStack` weitergereicht wird.

3.4 Beispiel: Vererbung mit gesetztem Parameter [26]

```
public class FormStack extends SimpleStack<Form> {

    public FormStack(int size) {
        super(size);
    }
```

```
    public Form peek() {
        Form wert = pop();
        push(wert);
        return wert;
    }
}
```

Hier wird der Typ-Parameter der Superklasse bei der Vererbung gesetzt, d. h. die Subklasse einer generischen Klasse ist selbst nicht mehr generisch. Der Typ-Parameter der Klasse `SimpleStack` wird im Rahmen der `extends`-Klausel gesetzt.

3.5 Generics als Compile-Zeit-Konstrukt [28]

Java Generics existieren nur zur Compile-Zeit

- **Typlöschung** (type erasure): Typ-Informationen werden entfernt — Typlöschung
  - Typ-Parameter ist zur Laufzeit vom Typ `Object`
  - Typ-Parameter kann nicht in statischen Variablen oder Methoden verwendet werden
  - Typ-Parameter kann nicht verwendet werden, um Objekte zu erzeugen
- Es gibt nur eine einzige Klasse pro generischem Typ, den **Raw type** — Raw type
- Sie verhalten sich damit grundlegend anders als C++-Templates

Die Besonderheit der Generics in Java ist, dass der Typ-Parameter nur zur Compilezeit existiert. Im generierten Byte-Code gibt es ihn nicht mehr, weswegen man von *Typlöschung* spricht. Der ursprünglich beliebig wählbare Typ-Parameter wird im Bytecode einfach durch `Object` ersetzt, sodass zur Laufzeit der generische Typ gar nicht mehr generisch ist. Von der Klasse existiert nur eine einzige Ausprägung, der *raw type*. Da aber der Compiler alle Verwendungen des Typs bereits zur Compilezeit überprüft und wo nötig Casts einfügt, ist die Typlöschung für den Verwender nahezu unsichtbar.

Der Grund für dieses ungewöhnliche Vorgehen liegt darin, dass die Java-Entwickler die Kompatibilität zu existierenden Java-Programmen nicht zerstören wollten. Vor allem sollten Programme, die Collection-Klassen aus historischen Gründen ohne Generics benutzen, weiter funktionieren.

Aus der Typlöschung folgen einige Konsequenzen, die die Programmierung mit Generics nicht unbedingt vereinfachen:

- Man kann keine Objekte mit `new T()` vom Typ des Typ-Parameters erzeugen.
- Man kann keine Arrays mit `new T[]` vom Typ des Typ-Parameters erzeugen.
- Man kann den Typ-Parameter nicht in statischen Methoden verwenden.

An dieser Stelle wird sofort klar, warum der Ausdruck `stack = (T[])new Object[size]` mit `T[] stack` aus dem ersten Beispiel zu Generics funktioniert:

- Man muss ein Object-Array anlegen, weil ein `new T[size]` nicht zulässig ist.
- Zur Compilezeit muss gecastet werden, weil `stack` vom Typ `T[]` ist und sonst die Zuweisung an ein Object-Array nicht durchgehen würde. Zur Laufzeit müsste der Cast aber eigentlich fehlschlagen.
- Nach dem Kompilieren wird im Bytecode aus `T[] stack` wegen der Type-Erasure `Object[] stack` und aus `(T[]) new Object[size]` `(Object[])new Object[size]` Somit ist durch die Typ-Löschung der eigentlich falsche Ausdruck wieder richtig.

Bei C++-Templates werden generische Typen über Code-Generatoren realisiert, d. h. für jede Ausprägung eines Templates wird der Code für eine Klasse bzw. Funktion generiert und compiliert. Im Gegensatz zu Java existieren C++-Templates also während der Laufzeit und es gibt ein Klasse pro Typ-Parameter-Wert.

3.6 Beispiel: Type Erasure [29]

```
var s1 = new SimpleStack<String>(4);
var s2 = new SimpleStack<Integer>(4);
SimpleStack sRaw = new SimpleStack(4);

System.out.println(s1.getClass());
System.out.println(s2.getClass());
System.out.println(sRaw.getClass());
```

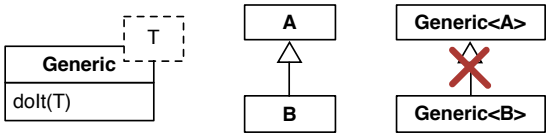
```
class pr2.erasure.SimpleStack
class pr2.erasure.SimpleStack
class pr2.erasure.SimpleStack
```

Im Beispiel wird der Name der Klasse ausgegeben, die hinter den jeweiligen Objekten liegt. Man sieht, dass aufgrund der Typ-Löschung aus allen generischen Typen eine einzige Klasse `SimpleStack` erzeugt wird.

3.7 Invarianz [30]

Generics sind **invariant** — invariant

- Generische Typen von Subtypen sind selbst keine Subtypen (invariante Typen, keine **Kovarianz**) — Kovarianz
- Es gibt keine Polymorphie zwischen verschiedenen Ausprägungen desselben generischen Typs



Anders als man also erwarten würde, führt die Tatsache, dass Generics von Subtypen selbst nicht wieder Subtypen voneinander sind dazu, dass man für die Aussage „egal welcher Typ-Parameter für den generischen Typ gesetzt wurde, ich nehme alles“ nicht `GenerischerTyp<Object>` sagen kann.

Eine Referenz vom Typ `GenerischerTyp<Object>` kann also nicht auf ein Objekt vom Typ `GenerischerTyp<String>` zeigen, obwohl eine Referenz vom Typ `Object` auf ein `String`-Objekt zeigen könnte. Es gibt also keine Polymorphie, wie man sie auf den ersten Blick vielleicht erwarten würde.

3.8 Beispiel: Invarianz [31]

```
public class Generic<T> {

    public void doIt(T param) {
        // ...
    }
}


public class Verwender {

    public static void main(String[] args) {

        A a1 = new B();
        A[] a2 = new B[1];

        Generic<A> ga = new Generic<B>(); // FEHLER!!
    }
}
```

Das Beispiel zeigt den drastischen Unterschied zwischen normalen Typen, Arrays und generischen Typen. Während normale Referenzen polymorph sind (`A a1 = new B()`) und sich auch Arrays so verhalten (`A[] a2 = new B[1]`), führt dies bei Generics zu einem Fehler (`Generic<A> ga = new Generic<B>()`).

3.9 Generics vs. Arrays [32]

Generics und Arrays unterscheiden sich grundsätzlich und harmonieren im Allgemeinen schlecht

- Arrays sind **kovariant** (covariant) — kovariant
- Generics sind **invariant** (invariant) — invariant
- Arrays prüfen ihren Typ zur Laufzeit (reified)
- Generics prüfen ihren Typ zur Compilezeit (erasure)
- Es ist nicht möglich ein Array zu erstellen aus
  - einem generischen Typ (`List<E>[]`)
  - einem parametrierten Typ (`List<String>[]`)
  - einem Typ-Parameter (`E[]`)

Wegen der grundlegenden Unterschiede funktionieren Arrays und Generics nur schlecht zusammen. Arrays sind *kovariant* und führen daher die Prüfung zur Laufzeit durch. Generics sind *invariant* und verlagern alle Prüfungen auf die Compilezeit.

Bei Arrays gibt es Polymorphie auf zwei Ebenen:

1. In einem Array vom Typ einer Superklasse, kann man, wegen der Polymorphie, ebenfalls Objekte von Subklassen speichern. Z. B. kann man in einem `Object`-Array Objekte vom Typ `String` oder `Date` speichern. Bei einer Zuweisung wird zur Laufzeit überprüft, ob der dynamische Typ des Arrays zum Typ des hinzugefügten Objektes passt.
2. Die Referenz auf das Array kann selbst wieder polymorph sein. Eine Referenz vom Typ `Object[]` kann auf ein Array vom Typ `String[]` zeigen.

```
// Polymorphie des Arrays
Object[] o1 = new Object[10];
o1[0] = "Hallo";
o1[1] = new Date();

// Polymorphie der Referenz
Object[] o2 = new String[5];
o2[0] = "Welt";
```

Bei Generics entfällt die zweite Ebene: In einem generischen Typ, dessen Typ-Parameter vom Typ einer Superklasse ist, kann man wegen der Polymorphie,Objekte von Subklassen speichern. Z. B. kann man in einem `Generic<Object>` Objekte vom Typ `String` oder `Date` speichern.

```
// Polymorphie des generischen Typs
Generic<Object> go = new Generic<>();
go.put("Hallo");
go.put(new Date());
```

Wie man bei Generics doch eine polymorphe Referenz definieren kann, wird später noch im Rahmen von Wildcards erläutert.

3.10 Beispiel: Kovarianz vs. Invarianz [33]

```
A[] a1 = new A[10];
A[] a2 = new B[10];
B[] b1 = (B[]) a2;

ArrayList<A> l1 = new ArrayList<A>();
ArrayList<A> l2 = new ArrayList<B>(); // FEHLER!!
```



```
de/smits_net/pr2/covariance/ArrayExample.java:26: incompatible types
found   : java.util.ArrayList<de.smits_net.pr2.covariance.B>
required: java.util.ArrayList<de.smits_net.pr2.covariance.A>
        ArrayList<A> l2 = new ArrayList<B>();
                          ^
```



3.11 Beispiel: Reified vs. Erasure (Array) [34]

```
Object[] array = new String[10];
array[0] = "Hallo";
array[1] = new Integer(15); // FEHLER!!
```



```
Exception in thread "main" java.lang.ArrayStoreException: ↩
    java.lang.Integer
     at de.smits_net.pr2.covariance.Refied.main(Refied.java:11)
```



Im Beispiel sieht man, dass die Überprüfung des dynamischen Typs eines Arrays bei einer Zuweisung erst zur Laufzeit passiert. Für den Compiler ist die Zuweisung `array[1] = ↩ new Integer(15)` korrekt, weil er nur den statischen Typ der Array-Referenz (`Object[]`) betrachtet. Zur Laufzeit schlägt sie aber fehl, weil das Array tatsächlich den dynamischen Typ `String[]` hat.

3.12 Beispiel: Reified vs. Erasure (Generic) [35]

```
ArrayList<Object> list = new ArrayList<String>(); // FEHLER!!
list.add(new Integer(15));
```



```
de/smits_net/pr2/covariance/Refied.java:16: incompatible types
```



```
found   : java.util.ArrayList<java.lang.String>
required: java.util.ArrayList<java.lang.Object>
        ArrayList<Object> list = new ArrayList<String>();
                                 ^
```

Bei Generics wird eine fehlerhafte Zuweisung bereits vom Compiler erkannt. Da der statische und der dynamische Typ bei der Deklaration nicht auseinanderfallen können, ist zu jedem Zeitpunkt für den Compiler klar, welche Typen in dem generischen Typ gespeichert werden können.

Kapitel 4

Wildcards und gebundene Parameter

4.1 Wildcards [37]

Wildcards

- **Wildcards** <?> zeigen an, dass jede beliebige Ausprägung eines generischen Typs möglich ist
- <?> ist die Kurzform von <? `extends Object`>

Bound Wildcards

- **Bound Wildcards** <? `extends T`> bzw. <? `super T`> stellen sicher, dass nicht jede beliebige Ausprägung des generischen Typs, sondern nur bestimmte möglich sind
- Wildcards können nur bei der Deklaration von Parametern und Variablen auftauchen
- Bei der Objekterzeugung und der Deklaration von generischen Typen können sie nicht verwendet werden

Anders als man also erwarten würde, führt die Tatsache, dass Generics von Subtypen selbst nicht wieder Subtypen sind, dazu, dass man für die Aussage „egal welcher Typ-Parameter für den generischen Typ gesetzt wurde, ich nehme alles“ nicht `GenerischerTyp<Object>` nehmen kann, sondern man `GenerischerTyp<?>` verwenden muss. Hintergrund ist, dass `GenerischerTyp<Object>` eben nicht wie `Object` selbst die Superklasse aller Ausprägungen des generischen Typs `GenerischerTyp` ist.

Eine polymorphe Referenz, die in der Lage ist auf Generics mit unterschiedlicher Ausprägung des Typ-Parameters zu zeigen, kann man durch Verwendung des Wildcards <?> definieren. Da es sich um die Deklaration einer Referenz handelt, kann das Wildcard nicht bei der Objekterzeugung verwendet werden. Damit entspricht `Generic<?>` der universellen Referenz `Object`.

gebundene Wildcards

Um den vollen Umfang der Polymorphie nutzen zu können, muss man in der Lage sein, den Wildcard einzuschränken. Dies geschieht über **gebundene Wildcards** (bound wildcards). Hier kann man einen Wildcard bezüglich der Vererbungshierarchie einschränken, d. h. der referenzierte generische Typ muss bezüglich seines Typ-Parameters Kriterien erfüllen:

- ? `super X` der Typ-Parameter muss vom Typ `X` oder einer *Superklasse* von X sein
- ? `extends X` der Typ-Parameter muss vom Typ `X` oder einer *Subklasse* von X sein

? ist nur eine Kurzform für ? `extends Object`.

Besonders das `super` ist verwirrend, da man diese Konstruktion aus der normalen Polymorphie nicht kennt. Hier kann eine Referenz immer nur auf den Typ oder einen Supertyp zeigen, nicht aber umgekehrt. Warum man diese Besonderheit bei Generics benötigt, wird in den nächsten Abschnitten noch erläutert.

4.2 Beispiel: Wildcard [38]

```
public class StackPrinter {
    public void printStack(SimpleStack<?> stack) {
        for (int i = stack.getSize(); i >= 0; i--) {
            System.out.printf("%d %s%n", i, stack.pop());
        }
    }
}

var stack = new SimpleStack<String>(10);
stack.push("!");
stack.push("World");
stack.push("Hello");

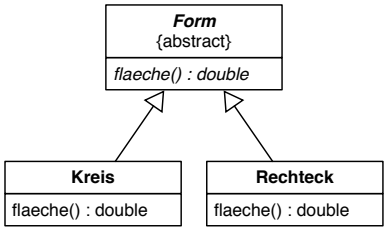
StackPrinter printer = new StackPrinter();
printer.printStack(stack);

SimpleStack<?> wildcard1 = new SimpleStack<String>(10);
SimpleStack<?> wildcard2 = new SimpleStack<?>(10); // FEHLER!!
```

Das Beispiel zeigt die Verwendung eines ungebundenen Wildcards. Die Methode `printStack` soll in der Lage sein, den Inhalt eines beliebigen Stacks auszugeben. Deswegen muss sie Stacks mit ganz unterschiedlichen Typ-Parametern akzeptieren, z. B. `SimpleStack<String>` oder `SimpleStack<Form>`. Dies kann man erreichen, indem man den Typ-Parameter als `SimpleStack<?>` deklariert, was eine Kurzform von `SimpleStack<? extends Object>` ist. Somit kann man alle Stacks übergeben, deren Typ-Parameter `Object` oder eine Subklasse davon ist. Da `Object` die Wurzel aller Vererbungshierarchien ist, kann man jeden `SimpleStack` übergeben.

Der Versuch ein Objekt von `SimpleStack<?>` zu erzeugen, schlägt fehl, weil das Wildcard nur bei Referenzvariablen, nicht aber bei der Objekterzeugung auftreten darf.

4.3 Beispiel: Bound Wildcard [39]




```java
public class Berechner {

    public double berechneFlaeche(SimpleStack<? extends Form> formen) {

        double summe = 0;

        for (int i = formen.getSize(); i >= 0; i--) {
            summe += formen.pop().flaeche();
        }

        return summe;
    }
}
```

Will man nur bestimmte Typ-Parameter zulassen, muss man den Wildcard einschränken. Dies passiert im vorliegenden Beispiel: Die Methode `berechneFlaeche` soll nur Stacks akzeptieren, die mindestens eine `Form` enthalten, weil nur `Form` eine Methode `flaeche()` hat. Aus diesem Grund wird das Wildcard auf `<? extends Form>` eingeschränkt. Jetzt kann die Methode nur mit Stacks aufgerufen werden, die den Typ-Parameter `Form` oder eine *Subklasse* davon haben, also im Beispiel `Form`, `Kreis` und `Rechteck`.

Da der Compiler jetzt weiß, dass immer mindestens ein `Form`-Objekt aus der `pop`-Methode kommt, kann man die `flaeche()`-Methode ohne Cast aufrufen.

Es ist typisch, dass man beim *Herausholen von Objekten* aus einem generischen Typ *extends* verwendet.

```java
public class SimpleStack<T> {
   ...
    public void popAll(Collection<? super T> collection) {
        while (pos > 0) {
            collection.add(pop());
        }
    }
}


var s = new SimpleStack<Form>(10);
```

```
s.push(new Rechteck(0.4, 0.4));
s.push(new Rechteck(0.7, 0.3));
s.push(new Kreis(0.4));
s.push(new Kreis(0.7));

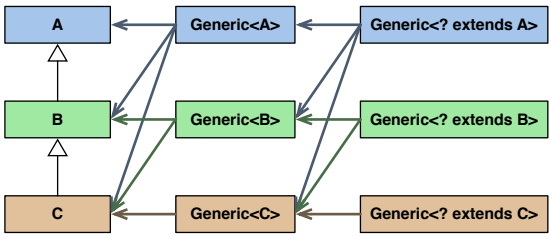
List<Object> l1 = new ArrayList<>();
s.popAll(l1);
```

In diesem Beispiel sollen alle Objekte eines Stacks in eine andere generische Klasse umkopiert werden. Aufgrund der Polymorphie muss man jetzt beim Wildcard `super T` verwenden, da es andernfalls vorkommen könnte, dass die falsche `Collection` übergeben würde.

Man kann dies wie folgt erklären: Wenn der `SimpleStack` den Typ-Parameter `B` hat, dann können darin Objekt vom Typ `B` oder einer Subklasse von `B` gespeichert werden (z. B. `C`, `D`, `E`, …). Will man diese Objekte umkopieren, dann muss die aufnehmende Klasse alle Objekte, die potenziell in `SimpleStack` enthalten sind, aufnehmen können. Ihr Typ-Parameter kann also nur `B` oder `A` oder `Object` sein, also muss er dem Typ-Parameter von `SimpleStack` entsprechen oder eine *Superklasse* davon sein.

Es ist typisch, das man beim *Speichern von Objekten* in einem generischen Typ *super* verwendet.

4.4 Wildcard mit extends [42]



Die Grafik zeigt noch einmal, welche Klasse welche Objekte enthalten kann, bzw. welche Referenzen auf welche Typen zeigen können. So kann eine Referenz vom Typ `Generic<? extends B>` auf die generischen Klassen `Generic<B>` und `Generic<C>` zeigen. `Generic<B>` kann dann Objekte vom Typ `B` und `C` enthalten.

4.5 Beispiel: Wildcard mit extends [43]

```
A a = new A();
B b = new B();
C c = new C();

Generic<A> ga = new Generic<>();
Generic<B> gb = new Generic<>();
Generic<C> gc = new Generic<>();
```

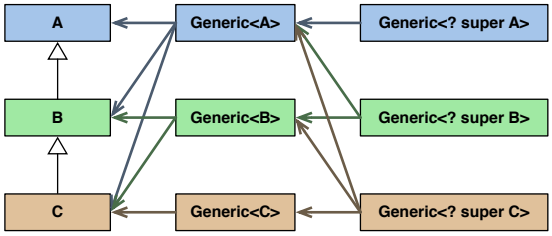
```
ga.add(a); ga.add(b); ga.add(c);
gb.add(b); gb.add(c);
gc.add(c);

Generic<? extends A> gea = ga;
gea = gb; gea = gc;

Generic<? extends B> geb = gb;
geb = gc;

Generic<? extends C> gec = gc;
```

4.6 Wildcard mit super [44]



Die Grafik zeigt noch einmal, welche Klasse welche Objekte enthalten kann bzw. welche Referenzen auf welche Typen zeigen können. So kann eine Referenz vom Typ `Generic<? super B>` auf die generische Klassen `Generic<B>` und `Generic<A>` zeigen. `Generic<B>` kann dann Objekte vom Typ `B` und `C` enthalten.

4.7 Beispiel: Wildcard mit super [45]

```
A a = new A();
B b = new B();
C c = new C();
Generic<A> ga = new Generic<>();
Generic<B> gb = new Generic<>();
Generic<C> gc = new Generic<>();

ga.add(a); ga.add(b); ga.add(c);
gb.add(b); gb.add(c);
gc.add(c);

Generic<? super A> gsa = ga;

Generic<? super B> gsb = gb;
gsb = ga;
```

```
Generic<? super C> gsc = gc;
gsc = gb; gsc = ga;
```

4.8 Beispiel: super und extends [46]



```
public void adder(Generic<? super B> g) {
    g.add(new C());
    g.add(new B());
    g.add(new A()); // FEHLER!! könnte ein Generic<B> sein
    B b = g.get(); // FEHLER!! könnte ein Generic<A> sein
}

public void getter(Generic<? extends B> g) {
    A a = g.get();
    B b = g.get();
    C c = g.get(); // FEHLER!! g könnte ein Generic<B> sein
    g.add(new B()); // FEHLER!! g könnte ein Generic<C> sein
}
```



```
public void doIt() {
    adder(new Generic<C>()); // FEHLER!! höchstens Generic<B> erlaubt
    adder(new Generic<B>());
    adder(new Generic<A>());

    getter(new Generic<C>());
    getter(new Generic<B>());
    getter(new Generic<A>()); // FEHLER!! mindestens Generic<B> benötigt
}
```

Man sieht an dem Beispiel deutlich, dass man beim Hinzufügen von Objekten zu einem generischen Typen als Wildcard `? super T` nehmen muss, wobei `T` der minimale Typ ist, den man hinzufügen kann – Subtypen von `T` kann man dann ebenfalls hinzufügen. Umgekehrt muss man beim Lesen von Daten aus dem Generic `? extends T` nehmen, wobei `T` der maximale Typ ist, den man als Rückgabetyp erwartet.

4.9 Gebundene Parameter [48]

Typ-Parameter können ebenfalls eingeschränkt werden

- Analog zu Wildcards können Typ-Parameter auf bestimmte (Unter-)Klassen beschränkt werden
- `<T extends K>` - Schränkt T auf K und Subklassen von K ein

4.10 Beispiel: Gebundene Parameter [49]

```
public class FormenStack<T extends Form> {

    ...

    public double flaeche() {

        double summe = 0;

        for (int i = getSize(); i >= 0; i--) {
            summe += pop().flaeche();
        }

        return summe;
    }
}
```

Das Beispiel zeigt einen gebundenen Typ-Parameter `T extends Form`. Bei der Klasse `FormenStack` kann man den Typ-Parameter `T` nur auf `Form` oder eine Subklasse davon (`Kreis`, `Rechteck`) setzen. Hierdurch ist jetzt garantiert, dass alle Objekte, die in dem Stack gespeichert werden, eine `flaeche`-Methode haben, die dann in der gleichnamigen Methode des `FormenStack` benutzt wird.

```
var s1 = new FormenStack<Kreis>(10);
s1.push(new Kreis(0.4));
s1.push(new Kreis(0.7));

var s2 = new FormenStack<Rechteck>(10);
s2.push(new Rechteck(0.4, 0.4));
s2.push(new Rechteck(0.7, 0.3));

var s3 = new FormenStack<Form>(10);
s3.push(new Rechteck(0.4, 0.4));
s3.push(new Rechteck(0.7, 0.3));
s3.push(new Kreis(0.4));
s3.push(new Kreis(0.7));

System.out.printf("Gesamtflaeche: %g%n", s1.flaeche());
System.out.printf("Gesamtflaeche: %g%n", s2.flaeche());
System.out.printf("Gesamtflaeche: %g%n", s3.flaeche());
```

Kapitel 5

Weitere Themen

5.1 Beispiel: Multiple Typ-Parameter [52]

```
public class Pair<T, V> {

    private T o1;
    private V o2;

    public Pair(T o1, V o2) {
        this.o1 = o1;
        this.o2 = o2;
    }

    public T getFirst() {
        return o1;
    }

    public V getSecond() {
        return o2;
    }
}
```

Die Anzahl der Typ-Parameter ist nicht beschränkt, man kann beliebig viele haben. Allerdings leidet die Übersichtlichkeit des Programms erheblich, wenn mehr als drei Typ-Parameter zum Einsatz kommen.

Neue Instanzen der Klasse `Pair` würde man dann z. B. mit folgendem Code erzeugen:

```
Pair<String, String> p1 = new Pair<>("Hello", "World");
Pair<Integer, String> geld = new Pair<>(100, "EUR");
```

5.2 Generische Methoden [53]

Auch Methoden können Typ-Parameter haben

- Sowohl statische als auch nicht-statische Methoden können als generische Methoden deklariert werden, z. B.
  `static <E> List<E> asList(E[] a)`
- Im Gegensatz zu Klassen muss der Verwender den Typ-Parameter nicht explizit setzen, der Compiler leitet ihn aus den Typen des Aufrufs ab: **Typ-Inferenz** (type inference)
- In seltenen Fällen muss man den Typ für die Methode explizit angeben

Typ-Inferenz

5.3 Beispiel: Generische Methoden [54]

```
public class Zusammenfasser {

    public static <T> SimpleStack<T> fasseZusammen(SimpleStack<T> s1, ↩
        SimpleStack<T> s2) {

        SimpleStack<T> ergebnis = new SimpleStack<T>(s1.getSize()
                + s2.getSize());

        for (int i = s1.getSize(); i >= 0; i--) {
            ergebnis.push(s1.pop());
        }

        for (int i = s2.getSize(); i >= 0; i--) {
            ergebnis.push(s2.pop());
        }

        return ergebnis;
    }
}
```

Bei **generischen Methoden** gibt es neben den normalen Parametern, die in den Klammern stehen noch Typ-Parameter. Diese werden, wie bei Klassen auch, in spitzen Klammern (`<T>`) notiert. Der Typ-Parameter wird, obwohl er ein Parameter ist, vor dem Rückgabetyp der Methode angegeben.

generischen Methoden

Im vorliegenden Beispiel muss man beachten, dass das erste `<T>` die Deklaration des Typ-Parameters ist, das zweite in `SimpleStack<T>` ist bereits eine Verwendung, genauso wie es sich bei `SimpleStack<T> s1` und `SimpleStack<T> s2` um Verwendungen handelt.

```
SimpleStack<String> stack1 = new SimpleStack<>(10);
stack1.push("Hello");
stack1.push("World");

SimpleStack<String> stack2 = new SimpleStack<>(10);
```

```
stack2.push("!");

SimpleStack<String> ergebnis = Zusammenfasser.fasseZusammen(stack1, stack2);
```

5.4 Beispiel: Generische Methoden (ohne Typ-Inferenz) [56]

```
public class Unifier {

    public <E> Set<E> unify(Set<? extends E> s1, Set<? extends E> s2) {
        Set<E> s = new HashSet<E>();
        s.addAll(s1);
        s.addAll(s2);
        return s;
    }
}
```



```
Set<Integer> s1 = new HashSet<>();
s1.add(1); s1.add(3);
Set<Double> s2 = new HashSet<>();
s2.add(10.0); s2.add(30.0);

Set<Number> union = new Unifier().<Number>unify(s1, s2);
System.out.println(union);
```



In diesem Beispiel muss der Typ für den Typ-Parameter `E` explizit angegeben werden. Der Hintergrund ist, dass der Compiler aus den Parametern der Methode den Typ nicht mehr ableiten kann, da die beiden Generics `s1` und `s2` unterschiedliche Typ-Parameter haben. Daher muss hier mit `<Number>` explizit angegeben werden, welchen Wert E annehmen soll.

Index

Bound Wildcards, 19

Containerklassen, 3

diamond, 8

Formalparameter, 7

gebundene Wildcards, 19
Generics, 6
generische Methoden, 6
generische Typen, 6
generischen Methoden, 27

invariant, 14, 16

kovariant, 16
Kovarianz, 14

Raw type, 13

Typ-Inferenz, 27
Typ-Parameter, 6
Typ-Variable, 6
Typlöschung, 13

Wildcards, 19

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Geschachtelte Klassen und Lambdas



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**
- 1.1 Fragen [5] . . . 1
- 1.2 Begriffe [6] . . . 1
- 1.3 Links zur Java-Insel [7] . . . 1
- 1.4 Überblick [8] . . . 2

2 Grundlagen **4**
- 2.1 Motivation für geschachtelte Klassen [13] . . . 4
- 2.2 Geschachtelte Klassen: Taxonomie [14] . . . 5
- 2.3 Geschachtelte Klasse [15] . . . 5
- 2.4 Gemeinsamkeiten [16] . . . 5

3 Static Member Classes **9**
- 3.1 Static Member Classes [24] . . . 9
- 3.2 Beispiel: Static Member Class [27] . . . 10
- 3.3 Zugriff auf private Bestandteile [28] . . . 10
- 3.4 Beispiel: Static Member Class [29] . . . 11
- 3.5 Zugriff auf private Bestandteile [30] . . . 11

4 Inner Classes **13**
- 4.1 Inner Classes [32] . . . 13

5 Nonstatic Member Classes **14**
- 5.1 Nicht-statische Elementklassen [35] . . . 14
- 5.2 Beispiel: Implizite Referenz [38] . . . 15
- 5.3 Beispiel: Zugriff auf implizites this [46] . . . 18
- 5.4 Beispiel: Erzeugen der Objekte [48] . . . 19

6 Local Classes **21**
- 6.1 Lokale Klassen [50] . . . 21
- 6.2 Beispiel: Lokale Klasse [53] . . . 22
- 6.3 Beispiel: Zugriff auf lokale Variablen [54] . . . 22
- 6.4 Beispiel: Zugriff auf Instanzvariablen [55] . . . 23
- 6.5 Beispiel: Ausbruch des Objekts [56] . . . 24

7 Anonymous Classes 26
7.1 Anonyme Klassen [59] . . . 26
7.2 Beispiel: Anonyme Klassen [62] . . . 27
7.3 Beispiel: Ausbruch des Objekts [63] . . . 28
7.4 Beispiel: Anonyme Klassen [64] . . . 28
7.5 Beispiel: Anonymer Konstruktor [66] . . . 30
7.6 Puzzler [68] . . . 31
7.7 Callbacks mit anonymen Klassen [70] . . . 31

8 Vererbung und geschachtelte Klassen 33
8.1 Geschachtelte Klassen und Vererbung [73] . . . 33
8.2 Beispiel: Innere Klassen und Vererbung [75] . . . 33
8.3 Quiz: Anonyme Innere Klassen [78] . . . 34

9 Lambdas 36
9.1 Das Problem [81] . . . 36
9.2 Lösung mit Lambda [84] . . . 37
9.3 Functional Interface [85] . . . 38
9.4 Syntax eines Lambdas [86] . . . 38
9.5 Vorgefertigte Interfaces [88] . . . 41
9.6 Beispiel: Vorgefertigte Interfaces [90] . . . 41
9.7 Capture [92] . . . 43
9.8 Lambdas als Rückgabewerte [93] . . . 44
9.9 Scoping bei Lambdas [94] . . . 45
9.10 Methodenreferenz [97] . . . 46

10 Realisierung innerer Klassen 48
10.1 Innere Klassen - Die Magie dahinter [100] . . . 48
10.2 Beispiel: Source Code [101] . . . 48
10.3 Beispiel: Echter Code für Klasse A [102] . . . 49
10.4 Beispiel: Echter Code für Klasse A$B [103] . . . 49

Index ii

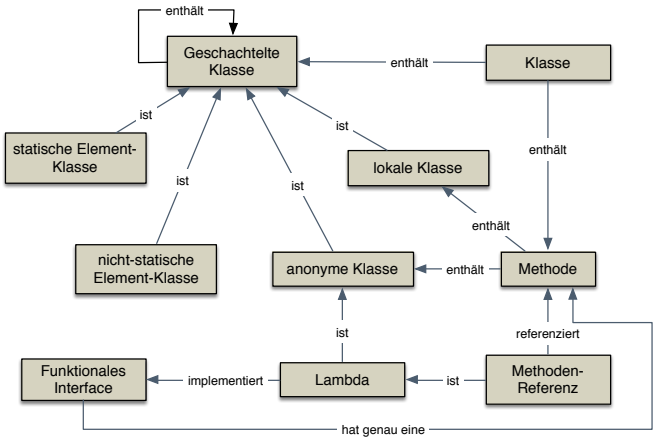
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie können Klassen in Klassen aufgenommen werden?
- Wie kann ich „mal eben“ eine Klasse und ein Objekt davon erzeugen?
- Wie kann ich Methoden herumreichen?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 10. Geschachtelte Typen
- 13. Lambda-Ausdrücke und funktionale Programmierung

1.4 Überblick [8]

Statisch geschachtelter Typ

```
class Outer {
    static class Inner {}
}
```



Ein Typ, innerhalb einer anderen Klasse, die mit dem Schlüsselwort `static` gekennzeichnet ist.

Innerer Typ / nicht-statisch

```
class Outer {
    class Inner {}
}
```



Ein Typ, innerhalb einer anderen Klasse, *ohne* das Schlüsselwort `static`.

Lokaler Typ

```
class Outer {
    void m() {
        class Inner {}
    }
}
```



Ein Typ innerhalb einer Methode.

Anonyme innere Klasse

```
class Outer {
    void m() {
        Runnable r = new Runnable() {
            public void run() {}
        };
    }
}
```



Eine unbenannte Klasse, innerhalb einer Methode, die von einer anderen, benannten Klasse, abgeleitet ist.

Lambda

```
class Outer {
    void m() {
        Runnable r = () -> {};
    }
}
```



Eine spezielle Konstruktion, die eine „freischwebende“ Methode erlaubt.

Kapitel 2

Grundlagen

2.1 Motivation für geschachtelte Klassen [13]

geschachtelte Klassen

Warum braucht man **geschachtelte Klassen**?

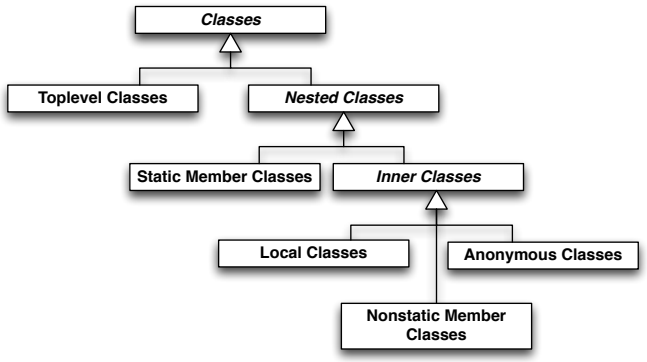
- Wohin mit Klassen, die nur eine interne Bedeutung haben?
- Wie übergibt man einfach einen Codeblock an eine Methode?
- Wie schreibe ich Event-Handler für eine grafische Benutzeroberfläche (GUI)?
- Wie stellt man eine Klasse zur Verfügung, die Parameter oder Rückgabewerte einer anderen Klasse repräsentiert?

Klassen sind in Java ein mächtiges Mittel zur Strukturierung von Programmen. Klassen können entweder für jeden sichtbar sein (`public`) oder nur innerhalb des Paketes benutzt werden (default). Was bisher vollkommen fehlt, ist die Möglichkeit Klassen zu definieren, die nur innerhalb einer anderen Klasse benutzt werden können. Diese Lücke wird von den *geschachtelten Klassen* geschlossen.

Eine der wichtigsten Anwendungszwecke der geschachtelten Klassen liegt bei der Programmierung von grafischen Oberflächen. Hier muss man sehr häufig eine Aktion mit einem Oberflächenelement (z. B. einem Knopf) verknüpfen. Diese Aktion liegt in Form einer Methode vor. Da man aber in Java keine Methoden übergeben kann, muss eine extra Klasse erzeugt werden, um die Methode über den Umweg eines Objektes übergeben zu können. Geschachtelte Klassen bieten hier eine erhebliche Erleichterung, da sie viel weniger Aufwand bei der Programmierung machen.

Mit Java 8 hat eine weitere Vereinfachung Einzug gehalten: durch sog. Lambda-Ausdrücke kann man Methoden noch einfacher als eigenständige Elemente handhaben und an andere Methoden übergeben. Lambdas werden hier aber erst nach den geschachtelten Klassen behandelt, weil sie viele Gemeinsamkeiten haben und historisch gesehen später eingeführt wurden.

2.2 Geschachtelte Klassen: Taxonomie [14]



Toplevel Classes

Die Taxonomie zeigt, dass es neben den bisher bekannten Klassen (**Toplevel Classes**) eine ganze Reihe von anderen Klassen gibt. Hierbei sind allerdings *nested* und *inner* classes keine real existierenden Klassen, sondern nur Gruppierungen von weiteren Arten von geschachtelten Klassen.

2.3 Geschachtelte Klasse [15]

geschachtelte Klassen

- Klassen in Klassen werden als **geschachtelte Klassen** (nested classes) bezeichnet
- Schachtelung hat *nichts* mit Vererbung zu tun
- Geschachtelte Klasse darf *nicht* so heißen wie umgebende Klasse

Immer wenn sich eine Klasse in einer anderen Klasse befindet, spricht man von *geschachtelten Klassen* (nested classes). Wichtig ist, dass die Beziehung zwischen umgebender und geschachtelter Klasse nichts mit Vererbung zu tun hat.

Die geschachtelten Klassen werden danach unterschieden, an welcher Stelle sie sich befinden:

- *innerhalb einer Klasse* befinden sich die *Static Member Classes* und die *Nonstatic Member Classes* und
- *innerhalb einer Methode* findet man die *Local Classes* und die *Anonymous Classes*.

Obwohl es einige Unterschiede zwischen den unterschiedlichen geschachtelten Klassen gibt, existieren auch viele Gemeinsamkeiten, die im Folgenden dargestellt werden sollen.

Eine wichtige Regel, die sich aus der internen Implementierung von geschachtelten Klassen ergibt, ist, dass sie nicht so heißen dürfen wie die umgebende Klasse.

2.4 Gemeinsamkeiten [16]

Klassen können *beliebig tief* ineinander geschachtelt werden

```
class A {
    class B {
```

```
        class C {
            class D {
            }
        }
    }
}
```

Es gibt keine Grenze für die Schachtelungstiefe von Klassen in Klassen. Allerdings hört man in der Praxis, aus Gründen der Übersichtlichkeit, normalerweise bei zwei Ebenen auf, d. h. in eine Toplevel Klasse wird normalerweise nur eine weitere Klasse geschachtelt.

Toplevel Klasse kann beliebige viele geschachtelte Klassen enthalten

Toplevel Klasse

```
class A {
    class B { }
    class C { }
    class D { }
}
```



Die Anzahl der Klassen, die in einer anderen Klasse enthalten sein können, ist nicht begrenzt.

Geschachtelte Klassen können auch *abstrakte Klassen*, *Interfaces*, *Enumerationen* oder *Records* sein

```
class A {
    abstract class B { }
    interface C { }
    class D { }
    enum E { K1, K2; }
}
```



Obwohl wir von geschachtelten Klassen sprechen, können Interfaces und Enumerationen in eine andere Klasse geschachtelt werden. Die geschachtelte Klasse kann abstrakt sein. Insofern können alle in Java denkbaren Typen in Klassen eingebettet werden.

Geschachtelte Klassen können voneinander oder anderen Klassen *erben*

```
class A {
    class B { }
    class C extends B { }
    class D<T> extends ArrayList<T> { }
    class E implements Serializable { }
}
```



Da es sich bei geschachtelten Klassen um weitgehend normale Klassen handelt, können sie Vererbungsbeziehungen untereinander oder zu anderen Klassen eingehen.

Eine geschachtelte Klasse kann eine *beliebige Sichtbarkeit* haben

```
class A {
    private class B { }
    class C { }
    protected class D { }
    public class E { }
}
```



Member

Geschachtelte Klassen sind genauso **Member** der umgebenden Klasse wie Attribute und Methoden. Daher können sie mit beliebiger Sichtbarkeit (`public`, `protected`, default, `private`) versehen werden. Hiervon ausgenommen sind nur die lokalen und anonymen Klassen, die gar keinen Sichtbarkeit tragen dürfen, da sie sich innerhalb von Methoden befinden und damit einen lokalen Scope haben (siehe unten).

Geschachtelte Klassen können auf *private Eigenschaften* der *umgebenden Klasse* zugreifen

```
class A {

    private int i = 12;

    class B {
        private int k = i;
    }
}
```



Eine Besonderheit von geschachtelten Klassen ist deren Fähigkeit, auf alle Eigenschaften der umgebenden Klasse zugreifen zu können, unabhängig von deren Sichtbarkeit. Deshalb können geschachtelte Klasse auf die privaten Attribute und Methoden der umgebenden Klasse zugreifen. Wenn man die bekannte Regel zur Hand nimmt, dass private Eigenschaften im gesamten Rumpf der Klasse sichtbar sind und sich überlegt, dass die geschachtelte Klasse sich ja genau in diesem Rumpf befindet, verliert dieser Zugriff seine Besonderheit.

Die umgebende Klasse kann auf *private Eigenschaften* der *geschachtelten Klasse* zugreifen

```
class C {

    C() {
        D d = new D();
        d.i = 100;
    }

    class D {
```



```
        private int i = 10;
    }
}
```

Analog zum Zugriff der geschachtelten Klasse auf die Attribute der umgebenden Klasse kann diese umgekehrt auf alle Eigenschaften und Methoden der geschachtelten Klasse zugreifen.

Kapitel 3

Static Member Classes

3.1 Static Member Classes [24]

Statische Element-Klasse (static member class)

Statische Element-Klasse

- Deklaration einer statischen Klasse innerhalb einer Klasse
- Hat Zugriff auf statische Variablen und statische Methoden der äußeren Klasse
- Hat *keinen* Zugriff auf Instanzvariablen und nicht-statische Methoden der äußeren Klasse

```
public class Outer {
    static class Inner { }
}
```



- Stellt nur eine besondere Form der Paketierung dar (Name der Klasse ist `Outer.Inner` statt nur `Inner`)
- Verhält sich ansonsten wie eine toplevel Klasse
- Objekte können einfach erzeugt werden
  `Outer.Inner ref = new Outer.Inner();`
- Können Instanzvariablen und nicht-statische Methoden enthalten
- Können statische Variablen und statische Methoden enthalten

Die einfachste Variante der statischen Klassen ist die *statsiche Element-Klasse*. Hierbei handelt es sich um eine geschachtelte Klasse, die mit dem Modifier `static` gekennzeichnet ist.

Die Besonderheit dieser Art von geschachtelter Klasse besteht darin, dass sie nur auf statische Eigenschaften der umgebenden Klasse zugreifen darf. Hier allerdings auch, wenn diese private sind. Der Hintergrund hierfür wird klarer, wenn später die nichtstatischen Element-Klassen eingeführt werden. An dieser Stelle kann man nur erklären, dass die statische Elementklasse ohne Objekt der äußeren Klasse existiert (sie ist statisch) und somit keine Möglichkeit hat, auf (nichtstatische) Attribute oder Methoden zuzugreifen.

Technisch gesehen führt die statische Elementklasse nur zu einer besonderen Form der Namensgebung: Durch das Schachteln muss der Namen der äußeren Klasse vor den Namen der

inneren gesetzt werden. Eine Klasse `B`, die statisch in eine Klasse `A` geschachtelt wurde, heißt dann also `A.B` statt nur `B`. Ansonsten verhält sie sich aber wie jede normale Klasse.

3.2 Beispiel: Static Member Class [27]

```
public class StaticMember {

    public static class InnereKlasse {
        public String toString() {
            return "Ich bin innen";
        }
    }

    public String toString() {
        return "Ich bin außen";
    }
}
```



```
StaticMember aeussere = new StaticMember();
StaticMember.InnereKlasse innere = new StaticMember.InnereKlasse();
System.out.println(aeussere); // -> "Ich bin außen"
System.out.println(innere); // -> "Ich bin innen"
```



Das Beispiel zeigt eine statische Element-Klasse `InnereKlasse`, die in die Klasse `StaticMember` geschachtelt wurde. Man sieht bei der Deklaration der Referenzvariable `innere` und der Objekterzeugung, dass der Name der geschachtelten Klasse tatsächlich aus dem Namen der äußeren und dem Namen der geschachtelten Klasse besteht.

3.3 Zugriff auf private Bestandteile [28]

```
public class PrivaterZugriff {
    private static int a = 5;
    static class InnereKlasse {
        private static int b = 7;
        private static void innen() {
            b = a;
            aussen2();
        }
    }

    private static void aussen1() {
        InnereKlasse.innen();
    }
```



```
    private static void aussen2() {
        InnereKlasse.b = 17;
    }
}
```

Dieses Beispiel zeigt, dass die geschachtelte Klasse problemlos auf das private, statische Attribut a der äußere Klasse zugreifen kann. Umgekehrt kann aber die äußere Klasse genauso die privaten Methoden und Attribute der geschachtelten Klasse benutzen.

3.4 Beispiel: Static Member Class [29]



```
public class Outer {
    private static final int CONSTANT = 15;

    public static class Inner {
        public static final double PI = 3.14159265;
        public int doSomething(int i) {
            return i * CONSTANT;
        }
    }
}

public class UseOuter {
    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner();
        inner.doSomething(15);
        double a = 2 * Outer.Inner.PI * 5.0 * 5.0;
    }
}
```

3.5 Zugriff auf private Bestandteile [30]



```
public class Zugriff {
    static class Inner1 {
        private int i;
        private void doIt(Inner2 inner2) {
            inner2.k++;
        }
    }
    static class Inner2 {
        private int k;
        private void doIt(Inner1 inner1) {
            inner1.i++;
        }
    }
```

```
    public void doIt(Inner1 i1, Inner2 i2) {
        i1.i = i2.k;
        i1.doIt(i2);
    }
}
```

Dieses Beispiel zeigt, dass man zum einen mehrere statische Element-Klassen in einer einzigen umgebenden Klasse haben kann und zum anderen, dass die statischen Element-Klassen untereinander auf die privaten Attribute der anderen zugreifen können. Da es sich in diesem Beispiel bei `i` und `k` nicht um statische Attribute handelt, ist allerdings ein Objekt vom Typ der Element-Klasse notwendig, das jeweils als Parameter an die Methoden übergeben wird.

Kapitel 4

Inner Classes

4.1 Inner Classes [32]

Innere Klasse (inner class) ist Überbegriff für

Innere Klasse

- Nichtstatische Elementklassen (nonstatic member classes)
- Lokale Klassen (local classes)
- Anonyme Klassen (anonymous classes)

Gemeinsamkeiten

- Durften bis Java 15 *keine* statischen Variablen oder Methoden enthalten
- Können `static final` Felder (Konstanten) enthalten
- Haben eine implizite Referenz auf ein Objekt der umgebenden Klasse (Details folgen)

Der Begriff *innere Klasse* bezeichnet eine Gruppe von geschachtelten Klassen, die sich dadurch auszeichnen, dass sie eine Beziehung zu einem Objekt der umgebenden Klasse haben. Diese Besonderheit wird noch detailliert erläutert.

Bis Java 16 durften die Klassen dieser Kategorie keine statischen Methoden oder Attribute enthalten. Ausgenommen hiervon waren nur Konstanten, also `static final` Attribute. Seit Java 16 ist diese Einschränkung gefallen.

Kapitel 5

Nonstatic Member Classes

5.1 Nicht-statische Elementklassen [35]

Nichtstatische Elementklassen (nonstatic member classes)

Nichtstatische Elementklassen

- Deklaration einer (nicht-statischen) Klasse innerhalb einer Klasse aber außerhalb einer Methode
- Hat Zugriff auf alle Attribute (statisch oder nicht-statisch) und Methoden (statisch oder nicht-statisch) der äußeren Klasse
- Name ist in der ganzen Klasse und abhängig von der Sichtbarkeit auch außerhalb bekannt

```
class A {
    class B { }
}
```



Der Zugriff auf die Methoden und Felder ist unabhängig von der Sichtbarkeit dieser. D. h. eine non-static member class kann auch auf private Methoden und Felder der äußeren Klasse zugreifen.

- Kann nur im Kontext eines Objekts der äußeren Klasse existieren
  `Outer outerObject = new Outer();`
  `Outer.Inner ref = outerObject.new Inner();`
- Hat eine implizite Referenz auf das Objekt der äußeren Klasse, das bei der Erzeugung benutzt wurde
  `Outer.this`

Auf den ersten Blick sieht die Forderung „die innere Klasse hat Zugriff auf die Attribute der umgebenden Klasse“ einfach aus. Wenn man sich aber vor Augen führt, dass die Attribute in einem *Objekt* der umgebenden Klasse gespeichert sind, kann der Zugriff nur dann möglich sein, wenn die innere Klasse das Objekt auch kennt. Sie benötigt also eine Referenz auf das Objekt der umgebenden Klasse, genauso wie Methoden eine `this`-Referenz benötigen, um auf die Daten des Objektes zuzugreifen. Über diese Referenz können dann die Methoden der inneren Klasse auf die Attribute zugreifen.

Würde man jetzt statische Methoden in der inneren Klasse erlauben, müsste sogar die innere Klasse selbst ein Objekt der äußeren Klasse kennen. Dies wäre aber eine paradoxe Konstruktion, denn welches der vielen Objekte, die von der äußeren Klasse im Laufe der Zeit angelegt werden, soll denn die innere Klasse erhalten? Da man diesen Widerspruch nicht auflösen kann, sind statische Methoden in inneren Klassen grundsätzlich verboten. Da statische Methoden aber verboten sind, wird die Konstruktion etwas einfacher.

Jedes Objekt der inneren Klasse besitzt eine implizite Referenz auf ein Objekt der äußeren Klasse. Über diese kann es dann in seinen nicht-statischen Methoden auf die Attribute des Objektes zugreifen. Der Zugriff auf die this-Referenz der äußeren Klasse erfolgt über die besondere Form `Outer.this` statt nur einfach `this`.

Als nächstes stellt sich die Frage, wie diese implizite Referenz entsteht. Die Lösung ist, dem Objekt der inneren Klasse bei seiner Erzeugung diese Referenz mitzugeben. Entweder indem man sie explizit dem `new`-Operator mitgibt (siehe nächstes Beispiel) oder weil die Erzeugung in einer Methode der äußeren Klasse erfolgt, wodurch automatisch klar ist, welches Objekt verwendet werden soll (die `this`-Referenz der Methode).

5.2 Beispiel: Implizite Referenz [38]

```
public class Computer {

    String hersteller = "HP";

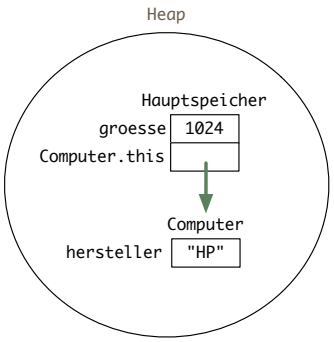
    class Hauptspeicher {
        int groesse = 1024;
    }
}
```



```
Computer computer = new Computer();
Computer.Hauptspeicher h = computer.new Hauptspeicher();
```



Um der inneren Klasse Zugriff auf die Daten eines Objektes der äußeren Klasse zu erlauben, werden die inneren Objekte immer im Kontext eines äußeren Objektes erzeugt. Man erkennt dies daran, dass das `new` auf einem Objekt der äußeren Klasse aufgerufen wird. Diese Referenz heißt in der inneren Klasse dann `<Name der auesseren Klasse>.this`, hier im Beispiel also `Computer.this`.



Im Bild erkennt man deutlich die Verknüpfung zwischen dem Objekt der inneren Klasse und dem Objekt der äußeren Klasse über die Referenz `Compuer.this`. Diese wurde nirgends deklariert, entsteht aber implizit durch die Beziehung zwischen innerer und äußerer Klasse bei der Erzeugung des inneren Objektes.

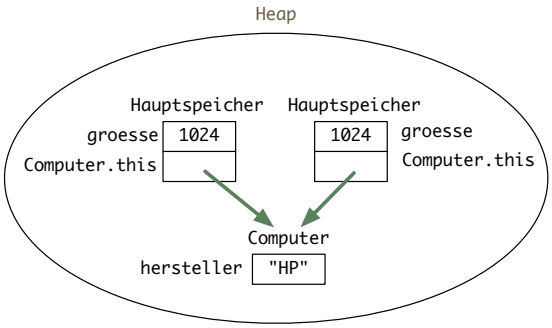
```
public class Computer {

    String hersteller = "HP";

    class Hauptspeicher {
        int groesse = 1024;
    }
}
```

```
Computer computer = new Computer();
Computer.Hauptspeicher h1 = computer.new Hauptspeicher();
Computer.Hauptspeicher h2 = computer.new Hauptspeicher();
```

Wenn mehrere Objekte der inneren Klasse im Kontext eines Objektes der äußeren Klasse erzeugt werden, dann teilen sie sich dieses Objekt, d. h. ihre `<outer>.this`-Referenzen zeigen auf dasselbe Objekt.



```
public class Computer {

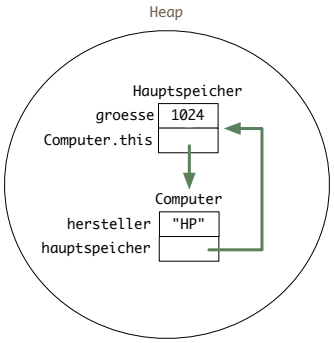
    String hersteller = "HP";
    Hauptspeicher speicher = new Hauptspeicher();

    class Hauptspeicher {
        int groesse = 1024;

        void printHersteller() {
            System.out.println(
                Computer.this.hersteller);
        }
    }
}
```

```
Computer computer = new Computer();
```

Wenn das Objekt der inneren Klasse innerhalb einer Methode oder eines Konstruktors der äußeren Klasse erzeugt wird, kann man sich die besondere Form des new-Operators sparen. Hier reicht ein einfaches `new`, da klar ist, in Kontext welchen Objektes die Instanz der inneren Klasse erzeugt wird. In diesem Beispiel ist das `new Hauptspeicher()` gleichbedeutend mit `this.new Hauptspeicher()`.




```
public class Computer {

    Hauptspeicher hauptspeicher = new Hauptspeicher();

    class Hauptspeicher {

        Speicherzelle zelle = new Speicherzelle();

        class Speicherzelle {

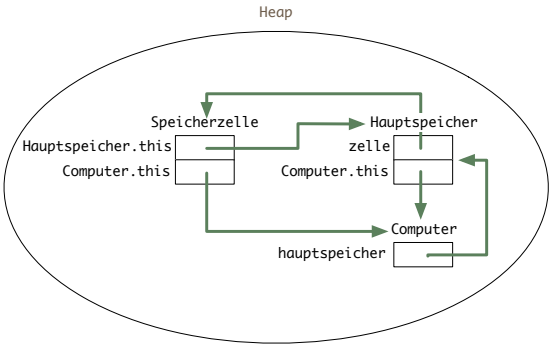
        }
```

```
    }
}
```

```
Computer computer = new Computer();
```

Wenn Klassen über mehrere Stufen ineinander geschachtelt werden, dann hat die innerste Klasse Zugriff auf die Instanzvariablen (und damit Objekte) aller sie umgebenden Klassen. Um dies zu gewährleisten, besitzt sie mehrere spezielle this-Referenzen und zwar eine pro Ebene. Diese Effekt kann man im Beispiel gut an der Klasse Speicherzelle sehen, die sowohl eine `Hauptspeicher.this`- als auch eine `Computer.this`-Referenz besitzt.

Es ist wichtig zu beachten, dass die Syntax für die this-Referenzen nicht hierarchisch ist, sondern flach: es heißt *nicht* `Computer.Hauptspeicher.this`, sondern nur `Hauptspeicher.this`. Hier wird jetzt auch klar, warum die inneren Klassen unterschiedliche Namen haben müssen. Andernfalls wäre unklar, welches Objekt gemeint ist.



5.3 Beispiel: Zugriff auf implizites this [46]

```
public class Computer {
    String hersteller = "HP";

    class Hauptspeicher {
        int groesse = 1024;

        class Speicherzelle {
            int kapazitaet = 10;

            void doIt() {
                System.out.println(Computer.this.hersteller);
                System.out.println(Hauptspeicher.this.groesse);
                System.out.println(kapazitaet);
            }
        }
```

```
    }
}
```

Hier ist noch einmal der Zugriff auf die speziellen this-Referenzen zu sehen und zwar in der qualifizierten Form.



```
public class Computer {

    String hersteller = "HP";

    class Hauptspeicher {
        int groesse = 1024;

        class Speicherzelle {
            int kapazitaet = 10;

            void doIt() {
                System.out.println(hersteller);
                System.out.println(groesse);
                System.out.println(kapazitaet);
            }
        }
    }
}
```

Wenn die Namen von Membern eindeutig sind und nicht überdeckt werden, kann man die spezielle this-Referenz weglassen und nur den Namen verwenden, da der Compiler problemlos feststellen kann woher die Variable stammt. Hier ist es also egal, ob man `hersteller` oder `Computer.this.hersteller` schreibt. Ersteres wird vom Compiler ohnehin in die zweit Form umgewandelt.

5.4 Beispiel: Erzeugen der Objekte [48]



```
Computer computer = new Computer();
Computer.Hauptspeicher speicher = computer.new Hauptspeicher();
Computer.Hauptspeicher.Speicherzelle zelle = speicher.new Speicherzelle();

Computer.Hauptspeicher.Speicherzelle zelle =
       new Computer().new Hauptspeicher().new Speicherzelle();
```

Will man die Objekte der inneren Klassen von außen erzeugen, muss man das Objekt der jeweils umgebenden Klasse beim `new` mit angeben.

Man sieht hier, dass zwar die Referenzen des inneren auf das äußere Objekt nicht hierarchisch sind, wohl aber die Namen der Klassen.

Kapitel 6

Local Classes

6.1 Lokale Klassen [50]

Lokale Klassen (local classes) Lokale Klassen

- Deklaration einer Klasse innerhalb einer Methode
- Name der Klasse ist nur innerhalb der Methode sichtbar
- Äußere Klasse kann nicht auf die lokale Klasse zugreifen
- Instanzen können nur innerhalb der Methode angelegt werden

```
class A {
    void methode() {
        class B { }
    }
}
```

Wenn man Klassen in Methoden deklariert, liegt eine *lokale Klasse* vor. Im Fall einer lokalen Klasse, kann die umgebende Klasse nicht mehr auf die Methoden und Attribute der inneren Klasse zugreifen, da der Name der Klasse außerhalb der Methode nicht bekannt ist. Dies ist analog zu einer lokalen Variablen, die man ebenfalls nicht von außen sehen kann.

Genauso wie die nichtstatischen Element-Klassen kann ein Objekt der lokalen Klasse nur im Kontext eines Objektes der umgebenden Klasse existieren. Wenn sie sich in einer normalen Methode befindet, ist dieses das Objekt, auf das die `this`-Referenz der Methode verweist.

Wenn die lokale Klasse in einer statischen Methode deklariert wird, kann sie nur noch auf statische Methoden und statische Felder der äußeren Klasse zugreifen, da sie dann nicht mehr im Kontext einer Instanz der äußeren Klasse existiert.

- Zugriff auf alle Attribute und Methoden der äußeren Klasse (Ausnahme lokale Klassen in statischen Methoden)
- Zugriff auf alle final Variablen der Methode, in der sie deklariert wurde
- Sichtbarkeit (`public`, `private`, `protected` oder default) kann *nicht* angegeben werden
- Deklaration muss vor Benutzung erfolgen

Für die lokale Klasse kann keine Sichtbarkeit angegeben werden, da sie bereits die geringst mögliche Sichtbarkeit hat: Sie ist nur innerhalb der Methode sichtbar und liegt damit sogar noch unter `private`.

6.2 Beispiel: Lokale Klasse [53]

```
public class LocalBeispiel {

    public static void main(String[] args) {

        class LocalClass {
            public String toString() {
                return "Ich bin lokal";
            }
        }

        LocalClass local = new LocalClass();
        System.out.println(local.toString());
    }
}
```

```
Ich bin lokal
```

Innerhalb der umgebenden Methode ist der Namen der Klasse bekannt und kann sowohl für Referenzvariablen als auch für die Erzeugung von Objekten verwendet werden. Außerhalb der Methode ist die Klasse nicht sichtbar.

Da bei lokalen Klassen die Deklaration vor der Benutzung erfolgen muss, kann man die Objekterzeugung im Beispiel nicht vor die Deklaration mit `class LocalClass {...}` ziehen.

6.3 Beispiel: Zugriff auf lokale Variablen [54]

```
public class LocalMitVariable {
    public static void main(String[] args) {

        final String ausgabe = "Ich bin lokal";

        class LocalClass {
            public String toString() {
                return ausgabe;
            }
        }

        LocalClass local = new LocalClass();
        System.out.println(local.toString());
```

```
    }
}
```

```
Ich bin lokal
```



Die lokale Klasse hat Zugriff auf finale lokale Variablen der umgebenden Methode. Dies ist auf den ersten Blick verwirrend, da lokale Variablen auf dem Stack liegen, Objekte aber auf dem Heap. Lokale Variablen werden vernichtet, sobald die Methode zurückkehrt. Da aber die Methode ein Objekt der lokalen Klasse zurückgeben kann, ist es möglich, dass das Objekt der lokalen Klasse länger lebt als die Methode selbst. Dies würde aber bedeuten, dass die lokalen Variablen vernichtet würden, obwohl das Objekt sie noch benötigt. In Programmiersprachen wie C, gilt es als schwerwiegender Programmierfehler Referenzen auf Variablen auf dem Stack außerhalb der Methode zu verwenden.

Java löst diese paradoxe Situation durch einen Kunstgriff: auf dem Stack können nur Referenzen und primitive Daten liegen. Durch die Bedingung, dass nur finale Variablen aus der lokalen Klasse genutzt werden können, wird gewährleistet, dass sie sich nach der Erzeugung des Objektes der lokalen Klasse nicht mehr ändern können. Wenn es Referenzen sind, liegt das referenzierte Objekt ohnehin auf dem Heap und lebt beliebig lang. Daher wird bei der Erzeugung des Objektes der lokalen Klasse einfach eine Kopie der Variablen in dem Objekt selbst gespeichert. Hierfür werden spezielle (unsichtbare) Attribute erzeugt. Diese Kopie ist dann bezüglich der Lebensdauer von der Methode entkoppelt, sodass das Problem gelöst ist. Das `final` garantiert, dass die Kopie und das Original der Variable sich nicht unterscheiden können.

6.4 Beispiel: Zugriff auf Instanzvariablen [55]

```
public class LocalMitVariableUndInstanz {
    private String variable = "Ich bin nicht lokal";

    public void doIt() {
        final String ausgabe = "Ich bin lokal";

        class LocalClass {
            public String toString() {
                return ausgabe + ", " + variable;
            }
        }
        LocalClass local = new LocalClass();
        System.out.println(local.toString());
    }
}
```





```
Ich bin lokal, Ich bin nicht lokal
```

Die lokale Klasse hat Zugriff auf Instanz-Variablen der umgebenden Klasse. Die Verwendung einer expliziten `this`-Referenz auf das Objekt der umgebenden Klasse ist hier nur nötig, wenn der Name der Variable innerhalb der Methode oder innerhalb der lokalen Klasse überdeckt wird.

6.5 Beispiel: Ausbruch des Objekts [56]

```
public class Escape {
    public static Object escape() {

        class Papillon {
            public String toString() {
                return "Ich bin geflohen";
            }
        }
        return new Papillon();
    }
}

Object o = Escape.escape();
System.out.println(o.toString());
```



```
Ich bin geflohen
```



Obwohl die lokale Klasse außerhalb der Methode nicht sichtbar ist, können Objekte dieser Klasse die Methode verlassen und zurückgegeben werden. Der dynamische Typ ist dann für den Außenstehenden nicht verwendbar (da nicht sichtbar), sondern er kann das Objekt nur über eine Referenz von einem Typ manipulieren, der in der Vererbungshierarchie über der lokalen Klasse steht und der ihm bekannt ist – im vorliegenden Beispiel `Object`.

```
public class Escape {
    public static Object escape() {

        final String text = "Ich bin geflohen";

        class Papillon {
            public String toString() {
                return text;
            }
        }
```



```
        return new Papillon();
    }
}
Object o = Escape.escape();
System.out.println(o.toString());
```

```
Ich bin geflohen
```

Kapitel 7

Anonymous Classes

7.1 Anonyme Klassen [59]

Anonyme Klasse (anonymous class) Anonyme Klasse

- Klasse hat keinen Namen und kann daher nicht noch einmal referenziert werden
- Deklaration einer Klasse *und* Erzeugung eines Objekts der Klasse innerhalb einer Methode
- Außerhalb der Methode nicht sichtbar
- Kann nur genau ein Interface implementieren *oder* genau eine Basisklasse ableiten, nicht beides gleichzeitig

```
class A {
    void methode() {
        Object o = new Object() { };
    }
}
```

Da anonyme innere Klassen keinen Namen haben, kann man keine Referenzen mit ihrem Typ definieren; die Referenzen haben immer den Typ der Klasse von der die anonyme Klasse abgeleitet wird oder des Interfaces, das von ihr implementiert wird. Instanziierung und Deklaration müssen zusammenfallen, da es später keine Möglichkeit gibt, mit `new` ein Objekt zu erzeugen – man kann den Namen der Klasse nicht angeben. Somit gibt es von einer anonymen inneren Klasse immer nur *genau ein Objekt*.

Wenn die anonyme Klasse in einer statischen Methode deklariert wird, kann sie nur noch auf statische Methoden und statische Felder der äußeren Klasse zugreifen, da sie dann nicht mehr im Kontext einer Instanz der äußeren Klasse existiert.

Aus diesem Grund ist es sinnlos, in anonymen inneren Klassen öffentliche Methoden zu definieren, die nicht bereits in der Oberklasse bzw. dem implementierten Interface vorhanden sind, da sie von außen über die Referenzen nicht erreichbar sind. Daher wird man im Allgemeinen nur Methoden der Superklasse überschreiben bzw. Methoden des Interfaces implementieren. Eigene öffentliche Methoden haben keinen Nutzen, weil für den Verwender nicht sichtbar.

Die Syntax zur Deklaration und gleichzeitigen Instanziierung einer anonymen Klasse ist:

```
Typ referenz = new Typ() {
  KLASSENDEFINITION
};
```

Als `Typ` kann nur entweder eine Basisklasse oder ein Interface angegeben werden. Beides gleichzeitig ist nicht möglich.

- Zugriff auf alle Attribute und Methoden der äußeren Klasse (Ausnahme anonyme Klassen in statischen Methoden)
- Zugriff auf alle final Variablen der Methode, in der sie deklariert wurde
- Sichtbarkeit (`public`, `private`, `protected` oder default) kann nicht angegeben werden

Für die anonyme Klassen (die den lokalen Klassen sehr ähnlich sind) kann keine Sichtbarkeit angegeben werden, da sie die geringstmögliche Sichtbarkeit bereits haben: sie ist nur im Augenblick ihrer Deklaration sichtbar.

Die Argumentation zu den lokalen Variablen wurde bereits bei den lokalen Klassen geliefert (s. o.).

7.2 Beispiel: Anonyme Klassen [62]

```
public class AnonymBeispiel {

    public static void main(String[] args) {

        Object o = new Object() {
            public String toString() {
                return "Anonym...";
            }
        };

        System.out.println(o);
    }
}
```

```
Anonym...
```

Die anonyme Klasse kann nicht referenziert werden. Daher fallen bei ihr die Deklaration und die Objekterzeugung zusammen, da man sich später nicht mehr auf die Klasse beziehen kann.

7.3 Beispiel: Ausbruch des Objekts [63]

```
public class Escape {

    public static Object escape() {

        return new Object() {
            public String toString() {
                return "Ich bin geflohen";
            }
        };
    }
}


Object o = Escape.escape();
System.out.println(o.toString());
```



```
Ich bin geflohen
```



Obwohl die anonyme innere Klasse nicht referenzierbar ist, können Objekte dieser Klasse die Methode verlassen und zurückgegeben werden. Der dynamische Typ ist dann für den Außenstehenden nicht verwendbar (da nicht sichtbar), sondern er kann das Objekt nur über eine Referenz von einem Typ manipulieren, der in der Vererbungshierarchie über der anonymen Klasse steht und der ihm bekannt ist – im vorliegenden Beispiel `Object`.

7.4 Beispiel: Anonyme Klassen [64]

```
public class Outer {

    public void sortSpecial(String[] strings) {

        Comparator<String> c = new Comparator<String>() {
            public int compare(String s1, String s2) {
                int result = s1.toUpperCase().compareTo(s2.toUpperCase());
                return result;
            }
        };

        Arrays.sort(strings, c);
    }
}
```



In diesem Beispiel wird eine anonyme Klasse dazu verwendet, einen Comparator, der Groß- und Kleinschreibung ignoriert, für die Sortierung eines String-Arrays zu verwenden.

Der hier dargestellte Code ist semantisch äquivalent mit dem folgenden (mit dem Unterschied, dass die Klasse hier einen Namen `DontCare` hat).

```
public class Outer2 {

    public void sortSpecial(String[] strings) {

        class DontCare implements Comparator<String> {

            public int compare(String s1, String s2) {
                int result = s1.toUpperCase().compareTo(s2.toUpperCase());
                return result;
            }
        }

        Comparator<String> c = new DontCare();

        Arrays.sort(strings, c);
    }
}
```

Noch kürzer könnte man das Beispiel schreiben als:

```
public class Outer {

    public void sortSpecial(String[] strings) {

        Arrays.sort(strings, new Comparator<String>() {

            public int compare(String s1, String s2) {
                int result = s1.toUpperCase().compareTo(s2.toUpperCase());
                return result;
            }
        });
    }
}
```

Mit den sogenannten *Lambdas*, die später noch eingeführt werden, würde es sich sogar verkürzen lassen zu:

```
public class Outer {

    public void sortSpecial(String[] strings) {
        Arrays.sort(strings, (s1, s2) -> ↩
            s1.toUpperCase().compareTo(s2.toUpperCase()));
    }
```

```
}
```

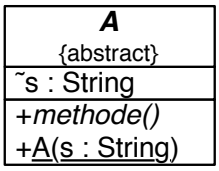
```
public class Outer {

    public void sortSpecial(String[] strings) {

        Arrays.sort(strings, new Comparator<String>() {

            public int compare(String s1, String s2) {
                int result = s1.toUpperCase().compareTo(s2.toUpperCase());
                return result;
            }
        });
    }
}
```

7.5 Beispiel: Anonymer Konstruktor [66]




```
public class A {
    String s;

    public A(String s) {
        this.s = s;
    }

    public void methode() {
        System.out.println(s);
    }
}
```

```
import java.util.Date;

public class Anonymous {
    public static void main(String[] args) {
        final String welt = "Welt";

        A a = new A("Hallo") {
            Date date = new Date();
```

```
            { // Konstruktor der anonymen Klasse
                this.s = this.s.toUpperCase() + " " + welt;
            }
            public void methode() {
                System.out.println(this.s + " am " + date);
            }
        };
        a.methode();
    }
}
```

Anonyme Klassen haben keinen Namen. Daher kann man keinen Konstruktor für sie angeben, denn der Konstruktor muss so heißen wie die Klasse. Es gibt aber die Möglichkeit mithilfe eines **Exemplar-Initialisierungsblocks** Initialisierungen für anonyme Klassen durchzuführen. Diese Blöcke können aber keine Parameter bekommen und keine Superklassenkonstruktoren explizit aufrufen.

Exemplar-Initialisierungsblocks

7.6 Puzzler [68]

```
import java.util.Date;

public class WasIstDennDas {

    public static void main(String[] args) {
        Date date = new Date() {{ setTime(400000000000L); }};
        System.out.println(date);
    }
}
```



```
Sat Sep 04 17:06:40 CEST 1982
```



Was man hier leicht übersehen kann, ist, dass gar kein Objekt vom Typ `Date` erzeugt wird, sondern eine anonyme Subklasse von `Date`. Im anonymen Konstruktor wird dann das Datum gesetzt, sodass nicht, wie man vielleicht erwarten würde das aktuelle Datum ausgegeben wird.

7.7 Callbacks mit anonymen Klassen [70]

```
public interface Funktion {
    public abstract int apply(int o1, int o2);
}
```





```
public class Berechnung {

    public static int berechne(int input1, int input2, Funktion funktion) {
        int ergebnis = funktion.apply(input1, input2);

        return ergebnis;
    }
}
```

```
public class TestBerechnung {

    public static void main(String[] args) {

        int ergebnis = Berechnung.berechne(7, 3, new Funktion() {
            public int apply(int o1, int o2) {
                return o1 - o2;
            }
        });

        ergebnis = Berechnung.berechne(5, 6, new Funktion() {
            public int apply(int o1, int o2) {
                return o1 + o2;
            }
        });
    }
}
```



Wir greifen hier noch einmal das Beispiel aus dem Kapitel „Objektorientierung“ auf und verwenden anonyme Klassen, um das Interface Funktion für Addition und Multiplikation zu implementieren.

Die Verwendung als Callback-Funktionen ist der Hauptanwendungszweck von anonymen inneren Klassen.

Kapitel 8

Vererbung und geschachtelte Klassen

8.1 Geschachtelte Klassen und Vererbung [73]

Geschachtelte Klassen können von der umgebenden Klasse erben

```
public class A {
    class B extends A { }
}
```

- Methoden und Attribute sind dann auf *zwei Wegen* sichtbar
  - Als *Member der äußeren Klassen*
  - Als *geerbte Attribute* und Methoden
- Die eigentlich getrennten Namensräume fallen zusammen
- Bei unqualifiziertem Zugriff haben die *ererbten Attribute Vorrang* vor den über die äußere Klasse sichtbaren
- Diese Konstruktion sollte vermieden werden, da sie sehr verwirrend und schwer zu lesen ist

Vererbungsbeziehungen

Geschachtelte Klassen können beliebige **Vererbungsbeziehungen** eingehen, auch zur umgebenden Klasse. Von einer solchen Konstruktion ist aber *dringend abzuraten*, da ein kompliziertes Gemenge zwischen geerbten und über die Schachtelung sichtbaren Attributen entsteht, die nur schwer zu beherrschen ist. Neben der Sichtbarkeit stellt sich auch die Frage, wo die Attribute eigentlich liegen: im Objekt der umgebenden Klasse oder als ererbte Attribute im eigenen Objekt. Auch hier ist Chaos vorprogrammiert.

8.2 Beispiel: Innere Klassen und Vererbung [75]

```
public class Outer {
    protected String s;
    private String t;
```

```
    protected void methode() { System.out.println("Outer:methode()"); }

    public class Inner extends Outer {
        protected void methode() { System.out.println("Inner:methode()"); }

        public void doIt() {
            s = "geerbter String";
            Outer.this.s = "String von aussen";
            t = "nicht über Vererbung sichtbar";
            Outer.this.methode();
            methode();
        }
    }
}
```

```
Outer o = new Outer();
Outer.Inner i = o.new Inner();
i.doIt();
System.out.println(o.s);
System.out.println(i.s);
```



```
Outer:methode()
Inner:methode()
String von aussen
geerbter String
```



8.3 Quiz: Anonyme Innere Klassen [78]

```
public class Twister {
    private final String name;
    Twister(String name) { this.name = name; }
    private String name() { return name; }
    private void reproduce() {
        new Twister("reproduce") {
            void printName() {
                System.out.println(name());
            }
        }.printName();
    }
    public static void main(String[] args) {
        new Twister("main").reproduce();
    }
}
```





```
main
```

Auf den ersten Blick würde man erwarten, dass das Programm nicht kompiliert, da die innere Klasse auf eine private Methode von `Twister` zugreift. Dies ist aber zulässig, weil innere Klassen Zugriff auf alle Methoden und Attribute der äußeren Klasse hat, unabhängig von der Sichtbarkeit. Private Methoden sind aber über die Vererbung nicht sichtbar (werden aber nicht vererbt). Deswegen greift der Aufruf `name()` in der inneren Klasse nicht auf eine entsprechende, geerbte Methode der inneren Klasse zu, sondern einfach auf die trotz `private` sichtbare Methode der äußeren Klasse.

Kapitel 9

Lambdas

9.1 Das Problem [81]

- Innere Klassen erlauben die Übergabe von Methoden an andere Methoden
- Übergabe erfolgt über den Umweg einer Klasse und eines Objektes
- Syntax ist umständlich und verwirrend
- Wieso kann man nicht einfach direkt eine Methode übergeben?

```
public interface Funktion {
    public int apply(int a, int b);
}
```



```
public class Berechnung {

    public static int berechne(int input1, int input2, Funktion funktion) {
        int ergebnis = funktion.apply(input1, input2);

        return ergebnis;
    }
}
```



```
int ergebnis = Berechnung.berechne(5, 7, new Funktion() {

    @Override
    public int apply(int a, int b) {
        return a + b;
    }
});

System.out.println(ergebnis);
```



Wenn man sich den Aufruf der Methode `berechne` mit der anonymen inneren Klasse ansieht, fällt sofort auf, dass trotz der bereits verkürzten Syntax noch extrem viel umständlicher Code geschrieben werden muss. Vor allem, wenn man sich überlegt, dass die Deklaration der Methode `apply` und das entsprechende Überschreiben der Methode in der anonymen inneren Klasse immer den gleichen Methodenkopf wiederholt.

Wünschenswert wäre eine Syntax, die es erlaubt diese ständigen Wiederholungen zu vermeiden und so den Quelltext übersichtlicher zu gestalten.

9.2 Lösung mit Lambda [84]

Seit Java 8 erlauben **Lambdas** eine verkürzte Definition

Lambdas

```
int ergebnis = Berechnung.berechne(5, 7, (x, y) -> x + y);
System.out.println(ergebnis);
```



```
Funktion add = (x, y) -> x + y;
Funktion sub = (x, y) -> x - y;

int e1 = add.apply(5, 7);
int e2 = sub.apply(10, 8);
```



Um noch einmal die deutliche Verkürzung der Definition durch Lambdas zu zeigen, hier das zweite Beispiel in klassischer (pre Java 8) Notation.

```
Funktion add = new Funktion() {
    @Override
    public int apply(int a, int b) {
        return a + b;
    }
};

Funktion sub = new Funktion() {
    @Override
    public int apply(int a, int b) {
        return a - b;
    }
};

int e1 = add.apply(5, 5);
int e2 = sub.apply(10, 8);
```



9.3 Functional Interface [85]

- Grundlage ist Interface mit einer einzigen abstrakten Methode
  - **SAM-Interface** (single abstract method interface)
  - **funktionales Interface** (functional interface)
- Annotation `@FunctionalInterface` verhindert Hinzufügen weiterer Methoden
- Compiler setzt Lambda in eine Implementierung des Interfaces um

SAM-Interface

funktionales Interface

`@FunctionalInterface`



```
@FunctionalInterface
public interface Aktion {
    public void run(String param);
}
```

Da Lambdas in Java nachträglich in die Sprache aufgenommen wurden und mit den vorhandenen Strukturen harmonieren müssen, konnte man nicht einfach hingehen und, wie z. B. in JavaScript, Funktionen als eigenständige Objekte definieren. Man muss daher Funktionen auf vorhandene Konstruktionen in Java abbilden. Da man bereits vor Java 8 schon Methoden über den Umweg eines Interfaces, einer Implementierung und eines entsprechenden Objektes übergeben konnte, hat man für Lambdas denselben Weg gewählt. Lambdas bieten also eigentlich nur eine einfache Syntax mit der man anonyme innere Klassen definieren kann. Allerdings wird eine Optimierung durchgeführt, die sich auf dem neuen ByteCode `invokedynamic` abstützt und es erlaubt, keine Klasse für das Lambda generieren zu müssen.

Grundlage eines Lambda-Ausdrucks ist ein Interface mit einer einzigen abstrakten Methode, das manchmal als *SAM-Interface* oder *funktionales Interface* bezeichnet wird. Um sicherzustellen, dass ein Interface genau eine abstrakte Methode hat, kann man durch die Annotation `@FunctionalInterface` den Compiler anweisen, dies zu überprüfen. Fügt man versehentlich eine weitere Methode hinzu (oder entfernt die einzig vorhandene), gibt es wegen der Annotation beim Kompilieren eine Fehlermeldung. Die Annotation sorgt also dafür, dass man nicht versehentlich in ein funktionales Interface eine weitere Methode einführt und damit seine Eigenschaft als funktionales Interface zerstört.

9.4 Syntax eines Lambdas [86]

- Lambda implementiert die Methode des funktionalen Interfaces
- Welches Interface implementiert wird, ergibt sich aus dem Kontext
  - Typ der Variable
  - Typ des Parameters
- Aufbau des Lambdas
  1. Parameter der Methode in Klammern (mit oder ohne Typ), durch Komma getrennt. Bei nur einem Parameter kann die Klammer entfallen
  2. Ein Pfeil ->

3. Rumpf der Methode. Bei nur einem Statement, kein Block { } nötig

- `return` ist nicht erforderlich, wenn nur ein Ausdruck im Lambda vorkommt

Wenn im Lambda nur ein Ausdruck vorkommt, ist dessen Ergebnis auch gleichzeitig der Rückgabewert des Lambdas.

Beispiele

- Ohne Typen: `(a, b) -> a + b`
- Mit Typen: `(int a, int b)-> a + b`
- Mit Block: `(a, b) -> { int s = a + b; return s; }`
- Ohne Klammer: `a -> a * a`
- Mit Klammer: `(a) -> a * a`
- Ohne Argumente: `() -> { System.out.println("Hallo"); }`

Der Java-Compiler muss wissen, zu welchem Interface ein Lambda gehört. Diese Information leitet er aus dem Kontext ab, in dem das Lambda vorkommt.

Wird es einer Variable zugewiesen, ergibt sich der Typ des Interfaces aus dem Typ der Variable.

```
Aktion a = s -> System.out.println(s);
```



Wird das Lambda an eine Methode übergeben, ergibt sich das funktionale Interface aus dem Typ des Parameters der Methode.

```
public static void executor(Aktion a) {
    a.run("Hallo");
}

public static void main(String[] args) {
    executor(s -> System.out.println(s));
}
```



Die Ableitung des Typs aus dem Kontext nennt man bei Lambdas **target typing**. Durch das Überladen von Methoden kann es passieren, dass mehrere Typen für das Lambda in Frage kommen. In diesem Fall bezieht der Compiler noch zusätzlich den Rückgabetyp der Methode und den des Lambdas mit in die Entscheidung ein, um herauszufinden, welchen Typ das Lambda haben soll. Das folgende Beispiel zeigt dies.

target typing

```
public interface Runner {
    public void run();
}
```





```
public interface Caller {
    public int call();
}
```

```
public class TargetTyping {
    public void doIt(Runner r) {
        r.run();
    }

    public void doIt(Caller c) {
        c.call();
    }

    public static void main(String[] args) {
        TargetTyping t = new TargetTyping();

        // implementiert Runner
        t.doIt(() -> { return; });

        // implementiert Caller
        t.doIt(() -> { return 1; });
    }
}
```

Syntaktisch besteht das Lambda aus zwei Teilen, die durch einen Pfeil -> getrennt werden.

Links vom Pfeil stehen die Parameter in Klammern und durch Komma getrennt. Für die Parameter kann der Typ angegeben werden. Dies ist aber unnötig, da sich der Typ aus der Methode des zu implementierenden Interfaces ergibt (**Typ-Inferenz**). Insofern wird man die Parameter im Normalfall weglassen. Gibt man für einen Parameter den Typ an, muss man es für alle Parameter tun.

Typ-Inferenz

Rechts vom Pfeil steht die Implementierung der Methode. Wenn diese nur aus einem einzigen Statement besteht, muss man keinen Block verwenden, sondern schreibt das Statement einzeln hin. Bei mehreren Statements steht der Rumpf des Lambdas in einem Block in geschweiften Klammern ({...}).

```
// Nur ein Statement, Block unnötig
Aktion a1 = s -> System.out.println(s);

// Mehrere Statements, Block
Aktion a2 = s -> {
    System.out.println(s);
    System.out.println("Nochmal" + s);
};
```

Kommt in dem Lambda nur ein einziger Ausdruck vor, so wird dieser zurückgegeben. Andernfalls muss man den Rückgabewert durch ein explizites `return` erzeugen.

9.5 Vorgefertigte Interfaces [88]

Im Paket `java.util.function` gibt es bereits eine Reihe von vorgefertigten funktionalen Interfaces

- `BiConsumer<T,U>` – Nimmt zwei Parameter vom Typ `T` und `U` und gibt kein Ergebnis zurück
- `BiFunction<T,U,R>` – Nimmt zwei Parameter vom Typ `T` und `U` und gibt ein Ergebnis vom Typ `R` zurück
- `BinaryOperator<T>` – Nimmt zwei Parameter vom Typ `T` und gibt ein Ergebnis vom Typ `T` zurück
- `Consumer<T>` – Nimmt einen Parameter vom Typ `T` und gibt kein Ergebnis zurück
- `Function<T,R>` – Nimmt einen Parameter vom Typ `T` und gibt ein Ergebnis vom Typ `R` zurück
- `Predicate<T>` – Nimmt einen Parameter vom Typ `T` und gibt einen Wahrheitswert zurück
- `Supplier<T>` – Nimmt keinen Parameter und gibt ein Ergebnis vom Typ `T` zurück
- …

Der Vorteil der bereits vorhandenen Interfaces ist, dass man sich häufig das Schreiben eigener funktionaler Interfaces ersparen kann, da für die meisten Anwendungsbereiche bereits entsprechende Interfaces vorgegeben sind. Da der Name der Methode im Lambda nicht auftaucht, ist die Verwendung der Interfaces aus `java.util.function` besonders einfach.

Für die primitiven Datentypen gibt es spezielle Interfaces, um das Autoboxing zu vermeiden:

- `BooleanSupplier`
- `DoubleBinaryOperator`, `DoubleConsumer`, `DoublePredicate`, `DoubleSupplier`, `DoubleToIntFunction`, `DoubleToLongFunction`, `DoubleUnaryOperator`
- `IntBinaryOperator`, `IntConsumer`, `IntPredicate`, `IntSupplier`, `IntToDoubleFunction`, `IntToLongFunction`, `IntUnaryOperator`
- `LongBinaryOperator`, `LongConsumer`, `LongPredicate`, `LongSupplier`, `LongToDoubleFunction`, `LongToIntFunction`, `LongUnaryOperator`

9.6 Beispiel: Vorgefertigte Interfaces [90]

```
import java.util.function.IntBinaryOperator;

public class Rechner {

    public static int berechne(int input1, int input2,
```

```
            IntBinaryOperator funktion) {
        int ergebnis = funktion.applyAsInt(input1, input2);
        return ergebnis;
    }

    public static void main(String[] args) {
        IntBinaryOperator sub = (a, b) -> a - b;
        IntBinaryOperator add = (a, b) -> a + b;

        System.out.println(berechne(5, 3, sub));
        System.out.println(berechne(1, 7, add));
    }
}
```

```
import java.util.function.IntBinaryOperator;

public class Rechner {

    public static int berechne(int input1, int input2,
            IntBinaryOperator funktion) {
        int ergebnis = funktion.applyAsInt(input1, input2);
        return ergebnis;
    }

    public static void main(String[] args) {
        System.out.println(berechne(5, 3, (a, b) -> a - b));
        System.out.println(berechne(1, 7, (a, b) -> a + b));
    }
}
```

Das Beispiel wurde bereits einmal im Kapitel zu Interfaces gezeigt, dort war die Implementierung aber deutlich umfangreicher und benötigte ein selbst geschriebenes Interfaces namens `Funktion`.

```
public interface Funktion {
    public abstract int apply(int o1, int o2);
}
```

```
public class Addition implements Funktion {

    public int apply(int o1, int o2) {
        return o1 + o2;
    }
}
```

```java
public class Subtraktion implements Funktion {

    public int apply(int o1, int o2) {
        return o1 - o2;
    }
}
```

```java
public class Berechnung {

    public static int berechne(int input1, int input2, Funktion funktion) {
        int ergebnis = funktion.apply(input1, input2);
        return ergebnis;
    }

    public static void main(String[] args) {
        Funktion sub = new Subtraktion();
        Funktion add = new Addition();

        System.out.println(berechne(5, 3, sub));
        System.out.println(berechne(1, 7, add));
    }
}
```

9.7 Capture [92]

- Lambda hat bei der Deklaration Zugriff auf die umgebenden Variablen
- Werte sind später beim Aufruf des Lambdas vorhanden
- Variablen dürfen nach Erzeugung des Lambdas nicht mehr verändert werden (am besten `final` machen) (muss **effectively final** sein)
- Lambdas mit Zugriff auf externe Variablen nennt man **capturing lambdas**

effectively final

capturing lambdas

```java
int minusEins = -1;

Comparator<String> cmp = (a, b) -> a.compareTo(b) * minusEins;
```

Das Lambda nimmt bei der Erzeugung die umgebenden Variablen mit, die aus dem Lambda heraus benutzt werden. Man kann es sich so vorstellen, dass es die **transitive Hülle** (transitive closure) der benutzen Daten bestimmt und behält. Aus diesem Grund nennt man im Englischen die Lambdas **Closures**.

transitive Hülle

Closures

Die Variable, die im Lambda verwendet wird, muss zwar nicht `final` deklariert werden, der Compiler überprüft aber, ob sie nach der Definition des Lambdas noch einmal verändert

wird. Passiert dies, wirft der Compiler eine Fehlermeldung. Daher spricht man in der Java-Spezifikation von Variablen, die **effectively final** sind. Prinzipiell könnte man die Variablen einfach `final` deklarieren, ohne dass sich etwas ändern würde, wie es bei lokalen Klassen gefordert wird. Der Vorteil, dass Lambdas dies nicht erzwingen ist, dass man vorhandenen Code nicht anpassen muss.

effectively final

```
int minusEins = -1;
Comparator<String> cmp = (a, b) -> a.compareTo(b) * minusEins;
minusEins = 2; // FEHLER!!
```

```
\Capture.java:10: error: local variables referenced from
a lambda expression must be final or effectively final
        Comparator<String> cmp = (a, b) -> a.compareTo(b) * minusEins;
                                                            ^
1 error
```

Aus den Regeln für den Zugriff auf Variablen aus dem Lambda folgt, dass Lambdas letztendlich *nicht Variablen, sondern Werte* mitnehmen.

9.8 Lambdas als Rückgabewerte [93]

Methoden können Lambdas zurückgeben

```
public class ComparatorFactory {

    public Comparator<String> createComparator() {
        return (a, b) -> a.compareTo(b);
    }
}
```

Dass man Lambdas als Rückgabewert verwenden kann, ist nicht verwunderlich, wenn man sich vor Augen führt, dass es sich einfach um eine verkürzte Schreibweise für die Implementierung von Interfaces und das Anlegen eines Objektes dieser Implementierung handelt. Würde man für das obenstehende Beispiel anonyme innere Klassen verwende, dann ergäbe sich dasselbe Ergebnis, nur der Code wäre länger.

```
public class ComparatorFactory {

    public Comparator<String> createComparator() {
        return new Comparator<String>() {

            @Override
            public int compare(String a, String b) {
```

```
                return a.compareTo(b);
            }
        };
    }
}
```

9.9 Scoping bei Lambdas [94]

- Lambdas sind eine verkürzte Schreibweise für anonyme innere Klassen
- Wichtiger Unterschied: `this` bezieht sich nicht auf das Objekt des Lambdas, sondern auf das umgebende Objekt

lexical scoping

Das Verhalten der Lambdas bezeichnet man als **lexical scoping**: Ein Bezeichner im Lambda bezieht sich direkt auf den umgebenden Kontext, wie man es im Programm sieht. Die Tatsache, dass das Lambda eigentlich ein Objekt ist, wird also nicht berücksichtigt. Daher bezieht sich `this` in einem Lambda nicht auf das (heimlich erzeugte) Lambda-Objekt, sondern das umgebende Objekt.

Dieses Verhalten ist insofern sehr sinnvoll, als dass die Lambda-Objekte nur deshalb erzeugt werden, weil die Funktionsweise von Java dies erfordert, da Methoden nur mithilfe von Objekten transportiert werden können. Durch das Scoping der Lambdas wird dafür gesorgt, dass sich dieses Implementierungsdetail nicht übermäßig in den Vordergrund drängt.

```
public class Scoping {

    public String toString() {
        return "Scoping";
    }

    public Runnable createRunner() {
        return () -> System.out.println(toString());
    }

    public static void main(String[] args) {
        new Scoping().createRunner().run();
    }
}
```

```
Scoping
```

Durch das Scoping bei Lambdas ist hier klar, dass mit `toString()` die entsprechende Methode der umgebenden Klasse gemeint ist. `toString()` ist nur eine verkürzte Schreibweise für `this.toString()` und aufgrund des Scopings von Lambdas bezieht sich `this` auf das Objekt von `Scoping` und nicht das (unsichtbare) Lambda-Objekt.

```java
public class Scoping {
    public String toString() {
        return "Scoping";
    }

    public Runnable createRunner() {
        return new Runnable() {
            public void run() { System.out.println(toString()); }
        };
    }

    public static void main(String[] args) {
        new Scoping().createRunner().run();
    }
}
```



```
de.smits_net.pr2.lambda.Scoping$1@15db9742
```



Verwendet man anstatt eines Lambdas eine anonyme innere Klasse, so bezieht sich `this` innerhalb der anonymen Klasse auf das eigene Objekt, sodass nicht die `toString`-Methode von `Scoping`, sondern die Methode vom anonymen Objekt verwendet wird. Da letzter nicht überschrieben wurde, kommt die geerbte Standardimplementierung von `Object` zum Zuge, die den Namen der Klasse und den Hash-Code ausgibt.

9.10 Methodenreferenz [97]

- Lambda gehört fest zu einem Interface
- Was tun, wenn man schon eine Methode hat, die passt aber das Interface nicht implementiert?
- Lösung: **Methodenreferenz** (method references) — Methodenreferenz
  - Statische Methoden: `Klasse::Methode`, z. B. `Integer::parseInt`
  - Nicht-statische Methoden: `Referenz::Methode`, z. B. `"Hallo"::toUpper`
  - Konstruktor: `Klasse::new`, z. B. `String::new`, `String[]::new`
- Kann nur in Zuweisung an eine Variable vom Typ eines Functional Interfaces benutzt werden

Die Methodenreferenz stellt eine einfache Brücke zwischen Lambdas und bereits vorhandenen Methoden dar. Wenn eine vorhandene Methode von der Struktur (Anzahl und Typ der

Parameter, Rückgabetyp) auf das funktionale Interface passt, kann man anstatt eines Lambdas einfach die Methodenreferenz verwenden.

```
public class Funktionssammlung {

    public static int add(int a, int b) {
        return a + b;
    }

    public static int sub(int a, int b) {
        return a - b;
    }
}
```



```
public class Main {
    public static void main(String[] args) {
        System.out.println(Funktion.rechne(5, 5, Funktionssammlung::add));
        System.out.println(Funktion.rechne(10, 8, Funktionssammlung::sub));
    }
}
```



Das Interface `Funktion` wurde bereits benutzt und sieht wie folgt aus:

```
public interface Funktion {
    public int apply(int a, int b);

    public static int rechne(int a, int b, Funktion fkt) {
        return fkt.apply(a, b);
    }
}
```



Kapitel 10

Realisierung innerer Klassen

10.1 Innere Klassen - Die Magie dahinter [100]

- Innere Klassen sind reine Compile-Zeit-Konstrukte
- In der Laufzeit (Java VM) existieren keine inneren Klassen
  - Der Compiler erzeugt besonderen Code, um innere Klassen auf normale top-level Klassen abzubilden
  - **synthetische Zugriffsmethoden**
  - **synthetische Felder** für die this-Referenzen
  - spezielle Konstruktoren
- Die inneren Klassen bekommen besondere Namen
  - `Outer$Inner` für benannte innere Klassen
  - `Outer$1`, `Outer$2` etc. für anonyme innere Klassen

synthetische Zugriffsmethoden

synthetische Felder

Man hat bei der Einführung der inneren Klassen davon abgesehen, die Virtual-Machine zu verändern und hat sie daher vollständig auf den Compiler abgestützt. Der Grund war wohl, dass man ansonsten die Kompatibilität zu allen existierenden VMs zerstört hätte und erhebliche Aufwände bei den VM-Anbietern entstanden wären. Diesen Ansatz wählt man bei Java relativ häufig, sodass andere Konstrukte wie z. B. die foreach-Schleife oder Enumerationen ohne Änderung der VM-Spezifikation eingeführt wurden.

Wegen der generierten synthetischen Methoden sind innere Klassen mit einem gewissen Geschwindigkeitsnachteil belegt, da ein einfacher Feldzugriff zum Aufruf einer statischen Methode werden kann. Moderne VMs optimieren zwar solche Konstrukte recht aggressiv (**Inlining**), trotzdem kann es vorkommen, dass die Geschwindigkeit leidet. In solchen Fällen sollte man die Methoden der inneren Klasse statt private einfach mit default-Sichtbarkeit versehen, da dann keine synthetischen Zugriffsmethoden mehr erzeugt werden müssen.

Inlining

10.2 Beispiel: Source Code [101]

```
public class A {
```

```
    private int i; private int j;
    public class B {
        private int i; private int j;

        private void methodB(String s) {
            i = A.this.i + A.this.j + j;
            methodA("Welt");
        }
    }

    private int methodA(String s) {
        B b = new B();
        b.i = 5;
        b.methodB("Hallo");
        return b.i + b.j;
    }
}
```

10.3 Beispiel: Echter Code für Klasse A [102]

```
public class A {

    private int i;
    private int j;

    public A() {}

    private int methodA(String s) {
        A$B b = new A$B(this);  // B b = new B();
        A$B.access$0(b, 5);     // b.i = 5;
        A$B.access$1(b, "Hallo"); // b.methodB("Hallo");
        return A$B.access$2(b) + A$B.access$3(b); // return b.i + b.j;
    }

    static int access$0(A a) { return a.i; }
    static int access$1(A a) { return a.j; }
    static int access$2(A a, String s) { return a.methodA(s); }
}
```

10.4 Beispiel: Echter Code für Klasse A$B [103]

```
public class A$B {
    private int i;
    private int j;

    private void methodB(String s) {
```

```
        // i = A.this.i + A.this.j + j;
        i = A.access$0(this$0) + A.access$1(this$0) + j;
        A.access$2(this$0, "Welt"); // methodA("Welt");
    }
    public A$B(A a) {
        this$0 = a;
    }
    final A this$0;
    static void access$0(A$B b, int i) { b.i = i; }
    static void access$1(A$B b, String s) { b.methodB(s); }
    static int access$2(A$B b) { return b.i; }
    static int access$3(A$B b) { return b.j; }
}
```

Index

Anonyme Klasse, 26

capturing lambdas, 43
Closures, 43

effectively final, 43, 44
Exemplar-Initialisierungsblocks, 31

funktionales Interface, 38

geschachtelte Klassen, 4, 5

Inlining, 48
Innere Klasse, 13

Lambdas, 37
lexical scoping, 45
Lokale Klassen, 21

Member, 7
Methodenreferenz, 46

Nichtstatische Elementklassen, 14

SAM-Interface, 38
Statische Element-Klasse, 9
synthetische Felder, 48
synthetische Zugriffsmethoden, 48

target typing, 39
Toplevel Classes, 5
Toplevel Klasse, 6
transitive Hülle, 43
Typ-Inferenz, 40

Vererbungsbeziehungen, 33

Wintersemester 2025/2026

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Datenstrukturen



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**
1.1 Links zur Java-Insel [5] . . . 1

2 Datentypen **2**
2.1 Abstrakter Datentyp [7] . . . 2
2.2 Liste [8] . . . 2
2.3 Stack [10] . . . 3
2.4 Queue [12] . . . 4
2.5 Ringpuffer [13] . . . 5

3 Listen **6**
3.1 Verkettete Liste [16] . . . 6
3.2 Einfach verkettete Liste [19] . . . 7
3.3 Einfach verkettete Liste: append [21] . . . 8
3.4 Einfach verkettete Liste: prepend [23] . . . 9
3.5 Einfach verkettete Liste [24] . . . 9
3.6 Doppelt verkettete Liste [25] . . . 10
3.7 Doppelt verkettete Liste: append [26] . . . 10
3.8 Doppelt verkettete Liste: prepend [28] . . . 11
3.9 Iterator [29] . . . 12

4 Hash-Tabellen **14**
4.1 Hash-Tabelle [32] . . . 14
4.2 Primitive Implementierung [35] . . . 15
4.3 Wahrscheinlichkeit für Kollisionen [38] . . . 16
4.4 Offenes Hashing [39] . . . 17

5 Bäume **19**
5.1 Baum [43] . . . 19
5.2 Binärbaum [45] . . . 20
5.3 Beispiel: Arithmetischer Ausdruck [46] . . . 20
5.4 Binärbaum [47] . . . 21
5.5 Binärbaum: Niveaus [48] . . . 21
5.6 Traversieren eines Baumes [49] . . . 22

6 Graphen **25**
6.1 Graphen [55] 25
6.2 Ungerichteter Graph [56] 25
6.3 Gerichteter Graph [58] 26
6.4 Gewichteter Graph [59] 26
6.5 Graph als dynamische Datenstruktur [61] 27

Index **ii**

Kapitel 1

Begriffe und Leitfragen

1.1 Links zur Java-Insel [5]

- 18. Einführung in Datenstrukturen und Algorithmen

Kapitel 2

Datentypen

2.1 Abstrakter Datentyp [7]

In der Informatik spielt die Verwendung von Datenstrukturen eine wichtige Rolle. Eine Datenstruktur ist eine spezielle Art von Datentyp, der es ermöglicht, Daten auf bestimmte Weise zu organisieren und zu verwalten. Abstrakte Datentypen (ADT) sind eine spezielle Form von Datenstrukturen, die sich durch ihre abstrakte und universelle Natur auszeichnen.

abstrakter Datentyp
ADT

abstrakter Datentyp (**ADT**): Beschreibung von Datenstrukturen unabhängig von ihrer späteren Implementierung in einer konkreten Programmiersprache

Konkrete Datentypen

- **Konkrete Datentypen** werden aus Basisdatentypen bzw. Java-Klassen konstruiert und sind somit direkt in einer Implementierung einsetzbar.

Abstrakte Datentypen

- **Abstrakte Datentypen** bilden nur eine Spezifikation der Schnittstelle nach außen, indem sie Operationen und ihre Funktionalität festlegen. Sie sind nicht direkt in ein Programm einbindbar.

Ein abstrakter Datentyp (ADT) ist eine spezielle Art von Datenstruktur, die nur über die Methoden und Operationen definiert wird, die auf ihr ausgeführt werden können, ohne ihre tatsächliche Implementierung zu offenbaren. Es ist eine abstrakte Vorstellung davon, wie Daten gespeichert und verarbeitet werden, ohne die tatsächlichen Details der Implementierung zu kennen.

Ein Beispiel für einen ADT ist eine Warteschlange (Queue). Eine Warteschlange ist eine Datenstruktur, bei der das erste Element, das hinzugefügt wurde, auch als erstes entfernt wird (First-In-First-Out, FIFO). Der ADT Warteschlange definiert die Methoden wie hinzufügen und entfernen von Elementen sowie Abfragen von Größe, aber nicht die tatsächliche Implementierung. Es kann implementiert werden, indem man ein Array oder eine Liste verwendet.

2.2 Liste [8]

Liste

Eine **Liste** ist eine abstrakte Datenstruktur, die eine *Folge von Elementen* speichert, wobei die Reihenfolge der Elemente signifikant ist. Listen ermöglichen es, Elemente einfach hinzuzufügen,

zu entfernen oder nach bestimmten Kriterien zu sortieren. Darüber hinaus können Listen in andere Datenstrukturen, wie z. B. Stacks oder Queues, umgewandelt werden.

Liste Liste

- Lineare Folge von Datenelementen
- Reihenfolge der Elemente wird erhalten
- Elemente können
  - hinzugefügt werden
  - entfernt werden
  - gesucht werden
- Grundlage für andere Datenstrukturen

Typische Operationen auf einer *Liste*

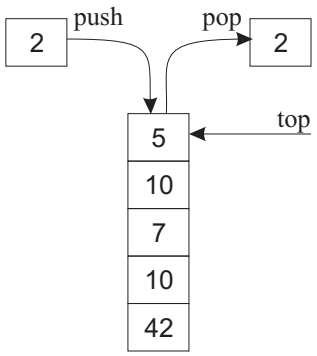
- `append`: Fügt ein Element am Ende hinzu append
- `insert`: Fügt ein Element am einer bestimmten Stelle hinzu insert
- `remove`: Entfernt ein Element remove
- `index`: Gibt den Index eines Elements zurück index
- `size`: Zählt die Anzahl eines Elements in der Liste size
- `clear`: Entfernt alle Elemente aus der Liste clear

2.3 Stack [10]

Ein **Stack** (**Stapel** oder **Kellerspeicher**) ist eine spezielle Art von Datenstruktur, die nach dem Last-In-First-Out (LIFO) Prinzip arbeitet. Das bedeutet, dass das letzte hinzugefügte Element als erstes entfernt wird. Stack Stapel Kellerspeicher

Stack auch **Kellerspeicher** oder **Stapel** Stack Kellerspeicher Stapel LIFO

- Last-In-First-Out (**LIFO**)
- Grundlage für viele Algorithmen
- Methodenaufruf in Programmiersprachen basiert auf Stack



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Typische Operationen auf einem **Stack** Stack

- `push`: fügt ein Element auf den Stack hinzu push
- `pop`: entfernt das oberste Element vom Stack und gibt es zurück pop
- `peek`: gibt das oberste Element des Stacks zurück, ohne es zu entfernen peek
- `isEmpty`: gibt an, ob der Stack leer ist oder nicht isEmpty

Ein konkretes Beispiel für die Verwendung eines Stacks ist das Auswerten von Ausdrücken in der Umgekehrten Polnischen Notation (RPN). In diesem Fall werden die Operanden (z. B. 2, 3) auf den Stack gelegt und die Operatoren (z. B. +, *) werden auf die Operanden angewendet, die sich auf dem Stack befinden.

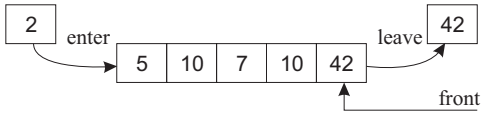
Ein Stack kann auch verwendet werden, um Funktionsaufrufe zu implementieren (Call Stack), und es ist auch Teil einiger Algorithmen wie Tiefensuche und Backtracking.

2.4 Queue [12]

Ein Queue (Warteschlange) ist ein abstrakter Datentyp, der nach dem First-In-First-Out (FIFO) Prinzip arbeitet. Das bedeutet, dass das erste hinzugefügte Element als erstes entfernt wird. Eine Queue besteht aus einer Liste von Elementen, die einen bestimmten Zugriff auf die Elemente ermöglicht.

Eine Queue hat typischerweise die folgenden Methoden:

- `enqueue`: fügt ein Element an das Ende der Queue hinzu. enqueue
- `dequeue`: entfernt das erste Element in der Queue und gibt es zurück. dequeue
- `peek`: gibt das erste Element in der Queue zurück, ohne es zu entfernen. peek
- `isEmpty`: gibt an, ob die Queue leer ist oder nicht. isEmpty



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

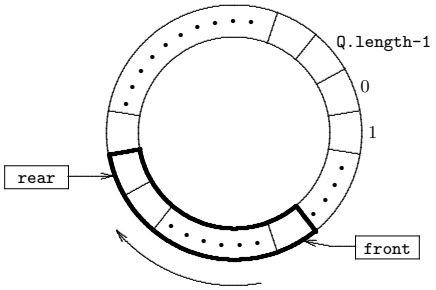
Ein konkretes Beispiel für die Verwendung einer Queue ist ein Drucksystem, in dem die Druckaufträge in der Reihenfolge ihres Eingangs verarbeitet werden. Ein weiteres Beispiel ist die Simulation von Warteschlangen in einem Computersystem, in dem Aufgaben in der Reihenfolge ihres Eingangs verarbeitet werden.

Queues können auf viele Arten implementiert werden, wie z. B. mit Arrays, Listen oder verketteten Listen. Es gibt auch spezielle Arten von Queues wie z. B. die **Priority Queue**, bei der die Elemente nach einem bestimmten Prioritätskriterium sortiert sind und die Dequeue-Operation das Element mit der höchsten Priorität entfernt. Priority Queue

2.5 Ringpuffer [13]

Ringpuffer

Ein **Ringpuffer** (auch bekannt als „circular buffer“ oder „circular queue“) ist eine spezielle Art von Datenstruktur, die auf einem Array basiert und die Eigenschaft besitzt, dass sie sich selbst überschreibt, wenn sie voll ist. Ein Ringpuffer hat eine feste Größe und ermöglicht es, Elemente hinzuzufügen und zu entfernen, ohne dass die Größe dynamisch verändert werden muss.



Quelle: Markus Nebel Sebastian Wild: Entwurf und Analyse von Algorithmen, 2. Auflage

Ein Ringpuffer hat typischerweise die folgenden Methoden:

- `put`: fügt ein Element am Ende des Ringpuffers hinzu. Wenn der Ringpuffer voll ist, wird das älteste Element überschrieben — put
- `get`: entfernt das älteste Element aus dem Ringpuffer und gibt es zurück — get
- `peek`: gibt das älteste Element des Ringpuffers zurück, ohne es zu entfernen — peek
- `isFull`: gibt an, ob der Ringpuffer voll ist oder nicht — isFull
- `isEmpty`: gibt an, ob der Ringpuffer leer ist oder nicht — isEmpty

Ein Anwendungsbeispiel für einen Ringpuffer ist die Aufzeichnung von Ereignissen in einer begrenzten Historie. Ein anderes Beispiel ist die Verarbeitung von Datenströmen, bei denen ältere Daten überschrieben werden, wenn kein Speicherplatz mehr verfügbar ist.

Kapitel 3

Listen

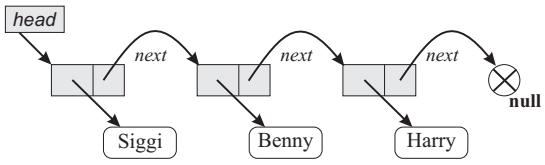
3.1 Verkettete Liste [16]

verkettete Liste

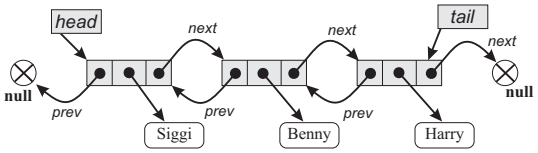
Eine **verkettete Liste** (linked list) stellt eine Datenstruktur dar, die aus einer Reihe von Knoten besteht, die miteinander verknüpft sind. Jeder Knoten enthält einen Wert und einen Verweis auf den nächsten Knoten in der Liste. Der letzte Knoten hat keinen Nachfolger und ist entsprechend gekennzeichnet, z. B. durch den Wert `null` in der Referenz auf das nächste Element.

verkettete Liste (linked list)

- zwei Formen
  - *einfach* verkettet
  - *doppelt* verkettet
- Knoten können leicht eingefügt und gelöscht werden
- Zugriff über Index ist aufwändig



Einfach verkettete Liste, Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage



Doppelt verkettete Liste, Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Typische Methoden

- `prepend`: Fügt ein Element vorne ein — prepend
- `append`: Fügt ein Element am Ende ein — append

- `insert`: Fügt ein Element an einer bestimmten Stelle ein — insert
- `remove`: Entfernt ein Element an einer bestimmten Stelle — remove
- `find`: findet ein Element — find
- `size`: gibt die Anzahl der Elemente in der Liste zurück — size
- `isEmpty`: gibt an, ob die Liste leer ist oder nicht — isEmpty

Eine verkettete Liste hat den Vorteil, dass Elemente einfach hinzugefügt oder entfernt werden können, ohne dass die restlichen Elemente verschoben werden müssen.

Es gibt zwei Arten von verketteten Listen: einfach verkettete Listen (engl. singly linked list) und doppelt verkettete Listen (engl. doubly linked list).

- Einfach verkettete Liste: Jeder Knoten enthält einen Wert und einen Verweis auf den nächsten Knoten.
- Doppelt verkettete Liste: Jeder Knoten enthält einen Wert, einen Verweis auf den nächsten Knoten und einen Verweis auf den vorherigen Knoten.

Die erste und die letzten Knoten haben spezielle Bezeichnungen: **head** und **tail**. — head, tail

verkettete Listen sind nützlich in Situationen, in denen häufig Elemente eingefügt oder entfernt werden, da diese Operationen in einer verketteten Liste in der Regel in konstanter Zeit durchgeführt werden können. Ein Nachteil von verketteten Listen ist, dass es schwieriger und zeitaufwendiger ist, ein Element an einer bestimmten Position zu finden, da man durch die Liste iterieren muss, um die richtige Position zu finden.

3.2 Einfach verkettete Liste [19]

Die **einfach verkettete Liste** besteht aus Knoten (`Node`), die neben den eigentlichen Daten (`data`) auch noch eine Referenz auf den nächsten Knoten (`next`) enthalten. — einfach verkettete Liste

Das folgende Beispiel zeigt eine einfach verkettete Liste, die `int`-Werte verarbeiten kann. Das Schema lässt sich auf jeden anderen Datentyp erweitern. In der Praxis würde man die Liste als *generichen Typ* ausprägen, sodass man mit einer Implementierung beliebige Datentypen speichern und verarbeiten kann.

Einzelner Knoten der Liste

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

Die Liste selbst hat nur eine einzige Referenz auf den Anfangsknoten (head).

Liste



```
public class LinkedList {
    private Node head;
    ...
}
```

Bei einer realen Implementierung würde man aus Effizienzgründen noch weitere Daten mitführen, z. B. die Anzahl der Elemente.

Eine Hilfsmethode `last()` innerhalb der Liste sucht den letzten Knoten der Liste. Diese Methode würde man bei einer doppelt verketteten Liste nicht benötigen, weil diese ihren Endknoten speichert.

Diese Methode muss die gesamte Liste durchlaufen, benötigt also bei einer Liste der Länge n auch n Schritte, $\mathcal{O}(n)$.

last



```
private Node last() {
    Node result = head;

    while (result != null && result.next != null) {
        result = result.next;
    }

    return result;
}
```

3.3 Einfach verkettete Liste: append [21]

Das folgende Beispiel zeigt, wie man am Ende der Liste einen neuen Eintrag anfügt. Die Methode `append()` greift hier auf die `last()`-Methode zurück, um das Ende der Liste zu finden.

append



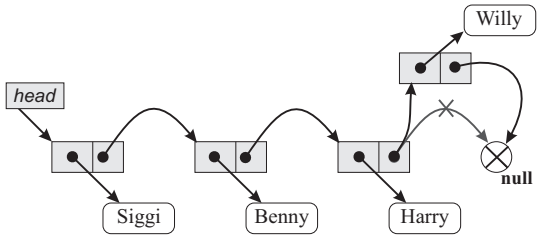
```
public void append(int data) {
    Node newNode = new Node(data);
    Node last = last();

    if (last == null) {
        head = newNode;
    }
    else {
        last.next = newNode;
```

```
    }
}
```

Die Benutzung der `last()`-Methode führt dazu, dass das Anfügen eines Elements bei der einfach verketteten Liste keine effiziente Operation ist.

Die folgende Grafik zeigt, wie das Programm die Liste beim Anfügen verändert.



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

3.4 Einfach verkettete Liste: prepend [23]

Die `prepend()`-Methode fügt ein Element am Anfang der Liste hinzu. Die Methode is effizient und unabhängig von der Größe der Liste, weil der erste Knoten bekannt ist und nicht gesucht werden muss.

prepend

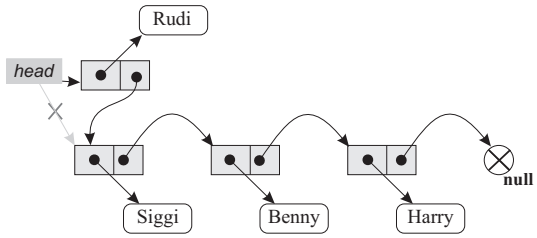
```
public void prepend(int data) {
    Node newNode = new Node(data);

    if (head == null) {
        head = newNode;
        return;
    }

    newNode.next = head;
    head = newNode;
}
```

3.5 Einfach verkettete Liste [24]

Die Grafik zeigt den Ablauf des Hinzufügens eines Elements am Anfang der Liste.



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

3.6 Doppelt verkettete Liste [25]

Das Anfügen von Elemente an eine Liste ist eine sehr häufige Operation in realen Programmen. Gerade hier ist aber die einfach verkettete Liste ineffizient und muss jedes Mal die gesamte Liste durchlaufen, um das Ende zu finden. Dieses Problem löst man mit der **doppelt verketteten Liste**, bei der jeder Knoten (`Node`) sowohl seinen Nachfolger (`next`), als auch seinen Vorgänger (`prev`) kennt.

doppelt verketteten Liste

Einzelner Knoten der Liste

```
class Node {
    int data;
    Node next;
    Node prev;
    Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}
```

Die Liste selbst speichert dann sowohl den ersten (`head`) als auch den letzten Knoten (`tail`) der Liste.

Liste

```
public class DoubleLinkedList {
    Node head;
    Node tail;
}
```

3.7 Doppelt verkettete Liste: append [26]

Durch die doppelte Verkettung ist es sehr effizient möglich, neue Knoten am Ende der Liste einzufügen. Der Aufwand dieser Operation ist unabhängig von der Anzahl der Elemente und damit in konstanter Zeit, $\mathcal{O}(1)$, möglich.

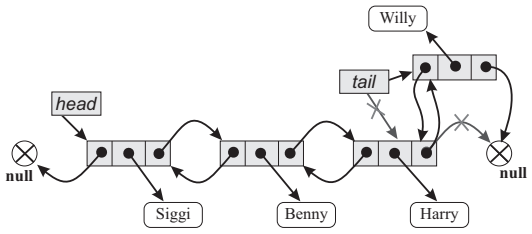
append

```
public void append(int data) {
    Node newNode = new Node(data);

    if (head == null) {
        head = newNode;
        tail = newNode;
        return;
    }

    tail.next = newNode;
    newNode.prev = tail;
    tail = newNode;
}
```

Die Grafik zeigt den Ablauf beim Anfügen eines neuen Elements an das Ende der List.



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

3.8 Doppelt verkettete Liste: prepend [28]

Möchte man ein Element am Anfang der doppelt verketteten Liste anfügen, so unterscheidet sich dieser Vorgang nicht von der einfach verketteten Variante. Der einzige Unterschied ist, dass man eine Referenz mehr (`prev`) setzen muss.

prepend

```
public void prepend(int data) {
    Node newNode = new Node(data);

    if (head == null) {
        head = newNode;
        tail = newNode;
        return;
    }

    newNode.next = head;
    head.prev = newNode;
    head = newNode;
}
```

3.9 Iterator [29]

Iterator
Zeiger
Cursor

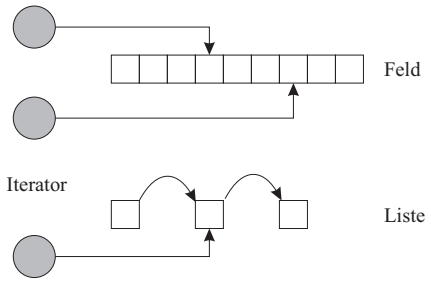
Ein **Iterator** (auch als **Zeiger** oder **Cursor** bezeichnet) ist ein abstrakter Datentyp, der es ermöglicht, durch die Elemente einer Datenstruktur (wie z. B. einer Liste oder eines Arrays) zu iterieren, ohne dass die tatsächliche Implementierung der Datenstruktur bekannt ist.

Iterator

Iterator bietet einheitlichen Zugriff auf unterschiedliche Datenstrukturen (Listen, Sets, …)

Typische Methoden

- `next`: gibt das nächste Element in der Datenstruktur zurück und bewegt den Iterator eine Position weiter
- `hasNext`: gibt an, ob es noch weitere Elemente in der Datenstruktur gibt
- `reset`: setzt den Iterator zurück auf den Anfang der Datenstruktur



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Ein Iterator ermöglicht es, durch die Elemente einer Datenstruktur zu iterieren, ohne dass die tatsächliche Implementierung der Datenstruktur bekannt ist. Es erhöht auch die Wiederverwendbarkeit von Code, da der Iterator unabhängig von der spezifischen Datenstruktur verwendet werden kann.

Beispiel: Ein Iterator kann verwendet werden, um durch die Elemente einer Liste zu iterieren, ohne dass die tatsächliche Implementierung der Liste (z. B. Array, verkettete Liste) bekannt ist. So kann der Iterator auf jede Art von Liste angewendet werden, solange sie die notwendigen Methoden zum Durchlaufen der Elemente bereitstellt.

Ein weiteres Beispiel wäre das Verwenden von einem Iterator für das Durchlaufen durch eine komplexe Datenstruktur wie einem Baum oder einem Graphen.

Iterator

Das folgende Beispiel zeigt einen einfachen `Iterator`, der über `int`-Array iterieren kann.

```
public class ArrayIterator {
    private final int[] array;
    private int pos = 0;

    public ArrayIterator(int[] array) {
```

```
        this.array = array.clone();
    }

    public boolean hasNext() {
        return pos < array.length;
    }

    public int next() {
        return array[pos++]; // ArrayIndexOutOfBoundsException
    }

    public void reset() { pos = 0; }
}
```

Die `clone()`-Operation im Konstruktor ist hier eingefügt, damit der Iterator gegen Änderungen des Arrays während der Iteration immun ist. Will man solche Änderungen live im Iterator sehen, muss man das `clone()` entfernen.

Kapitel 4

Hash-Tabellen

4.1 Hash-Tabelle [32]

Eine Hash-Tabelle ist eine abstrakte Datenstruktur, die verwendet wird, um Daten schnell zu speichern und abzurufen.

Hash-Tabelle (hash table) Hash-Tabelle

- Schneller Zugriff auf Daten $\mathcal{O}(1)$
- Normalerweise Schlüssel-Wert-Paare
- Vom Schlüssel wird Hash-Wert berechnet
- Hash-Wert bestimmt Position in der Tabelle

Es besteht aus Schlüssel-Wert-Paaren, wobei der Schlüssel verwendet wird, um einen bestimmten Wert schnell zu finden. Der Schlüssel wird durch einen Hash-Algorithmus in einen bestimmten Index im Speicher konvertiert, an dem der Wert gespeichert wird. Dieser Prozess wird als Hashing bezeichnet.

Beim Abrufen eines Werts wird der Schlüssel erneut durch den Hash-Algorithmus konvertiert und der Wert wird am entsprechenden Index im Speicher abgerufen. Dies ermöglicht eine schnelle Suche, da der Wert anhand des Schlüssels schnell gefunden werden kann, anstatt durch das Durchsuchen aller Daten.

Hash-Tabellen werden oft in Datenbanken und Cache-Systemen verwendet.

Hash-Tabellen können jedoch auch Kollisionen haben, sodass mehrere Schlüssel denselben Hashwert erzeugen und somit in dieselbe Stelle im Speicher gespeichert werden. Um dies zu lösen, gibt es verschiedene Techniken wie z. B. Open Addressing und Bucketing.

Ein weiteres Merkmal von Hash-Tabellen ist, dass die Größe der Tabelle im Voraus festgelegt werden muss. Dies kann dazu führen, dass die Tabelle ausgefüllt wird und es keinen Platz mehr für neue Einträge gibt. Um dies zu vermeiden, kann die Größe der Tabelle dynamisch angepasst werden, indem sie bei Bedarf vergrößert oder verkleinert wird. Dieser Prozess wird als Rehashing bezeichnet.

Hash-Tabellen sind sehr schnell bei der Suche, Einfügen und Löschen von Werten, da sie eine konstante Zeitkomplexität haben, normalerweise $\mathcal{O}(1)$ in durchschnittlichen Fällen.

Sie sind jedoch weniger effizient bei der Iteration über alle Elemente, da die Reihenfolge der Elemente in der Tabelle nicht unbedingt der Reihenfolge entspricht, in der sie hinzugefügt wurden.

Typische Methoden

- `add`: Fügt ein neues Element ein — add
- `contains`: Überprüft, ob ein Element bereits vorhanden ist — contains
- `remove`: Entfernt ein Element — remove

Index	Eintrag
0	
1	
2	42
3	
4	
5	
6	
7	
8	
9	119

Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

4.2 Primitive Implementierung [35]

Implementierung

- Array fester Länge (`m`) für die Elemente (**Buckets**) — Buckets
- Für Element wird ein Hash-Wert berechnet (`h`)
- Modulo-Operation bestimmt Index des Buckets
  `i = h % m`
- Es kann zu Kollisionen kommen (gleicher Index trotz unterschiedlichen Daten)

```java
public class SimpleHashTable {

    private static final int SIZE = 10;
    private final String[] buckets = new String[SIZE];

    public int idx(String s) {
        // hashCode kann < 0 liefern, wegen signed int
        return Math.abs(s.hashCode()) % SIZE;
    }

    public void add(String s) { buckets[idx(s)] = s; }

    public void remove(String s) { buckets[idx(s)] = null; }
```

```
    public boolean contains(String s) { return s.equals(buckets[idx(s)]); }
}
```

Diese Implementierung leidet unter Kollisionen, die vorkommen können und dann dazu führen, dass Elemente überschrieben werden.

Hier wird die Hash-Tabelle nicht wie üblich mit Schlüssel-Wert-Paaren realisiert, sondern die Buckets speichern nur einen einfachen Wert, einen String. Es ist aber relativ einfach, den String hier durch eine Datenstruktur zu ersetzen, die sowohl den Key, als auch den Value enthält. Der Index des Buckets bestimmt sich dann aus dem Hash-Code des Keys.

Das folgende Beispiel zeigt das Problem der Kollisionen.

```
SimpleHashTable sh = new SimpleHashTable();
sh.add("Januar");
sh.add("Februar");
sh.add("März");
sh.add("April");
sh.add("Mai");

System.out.println(sh.contains("Februar")); // -> False
System.out.println(sh.contains("Januar")); // -> True
```

Die Tabelle sieht nach diesem Code-Fragment wie folgt aus:

Inhalt der Tabelle

```
[0]:
[1]: März
[2]:
[3]:
[4]: April
[5]:
[6]:
[7]: Januar
[8]:
[9]: Mai
```

4.3 Wahrscheinlichkeit für Kollisionen [38]

Die Wahrscheinlichkeit, `n` Elemente in eine Hash-Tabelle der Kapazität `m` ohne Kollision einzufügen ist $(1 \leq n \leq m)$:

$$P = 1 - \frac{m!}{(m-n)! \cdot m^n}$$

Im vorliegenden Beispiel ergibt sich bei `m = 10` und `n = 5` eine Wahrscheinlichkeit von

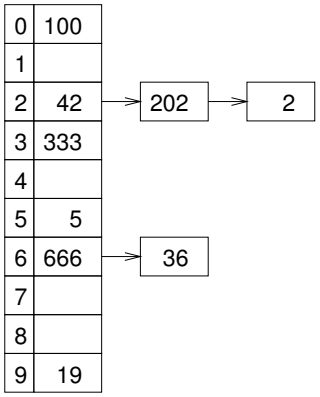
$$P = 1 - \frac{10!}{(10-5)! \cdot 10^5} = 0,6976 = 69\%$$

4.4 Offenes Hashing [39]

Offenes Hashing verhindert Überschreiben:

Offenes Hashing

- Buckets enthalten lineare Listen
- Bei Kollisionen werden die Elemente in die Liste eingetragen



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Um Überläufe handhaben zu können, kann in einer Hash-Tabelle jeder Bucket durch eine verkettete Liste realisiert werden. Elemente werden an den Anfang der Liste angefügt bzw. darin gesucht oder daraus gelöscht. Eine leere Liste entspricht einem nicht belegten Bucket. Dieses Verfahren wird als offenes Hashing bezeichnet, da prinzipiell beliebig viele Überläufe stattfinden können.

Eine (immer noch sehr primitive) Implementierung des offenen Hashings könnte wie folgt aussehen:

```
public class BetterHashTable {

    private static final int SIZE = 10;
    private final LinkedList[] buckets = new LinkedList[SIZE];

    public BetterHashTable() {
        for (int i = 0; i < SIZE; i++) {
            buckets[i] = new LinkedList();
        }
    }
    public int idx(String s) { return Math.abs(s.hashCode()) % SIZE; }
```

```
    public void add(String s) { buckets[idx(s)].add(s); }

    public void remove(String s) { buckets[idx(s)].remove(s); }

    public boolean contains(String s) { return buckets[idx(s)].contains(s); }
}
```

Die Verwendung der `LinkedList` ist nicht ganz unproblematisch, weil hier ein Array von einem generischen Datentyp erzeugt wird, was zu Code führt, der nicht typsicher ist. Dieses Problem soll hier aber nicht weiter diskutiert werden und es sei auf das Kapitel zu generischen Typen verwiesen.

Verwendet man diese neue Implementierung, geht der Februar nicht mehr verloren.

```
BetterHashTable sh = new BetterHashTable();
sh.add("Januar");
sh.add("Februar");
sh.add("März");
sh.add("April");
sh.add("Mai");

System.out.println(sh.contains("Februar")); // -> true
System.out.println(sh.contains("Januar")); // -> true
```

Die Tabelle sieht nach diesem Code-Fragment wie folgt aus:

Inhalt der Tabelle

```
[0]: []
[1]: [Februar, März]
[2]: []
[3]: []
[4]: [April]
[5]: []
[6]: []
[7]: [Januar]
[8]: []
[9]: [Mai]
```

Kapitel 5

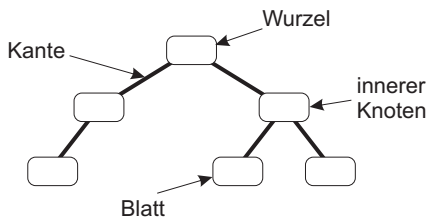
Bäume

5.1 Baum [43]

Ein Baum ist ein abstrakter Datentyp, der eine hierarchische Struktur darstellt, die aus **Knoten** (nodes) und **Kanten** (edges) zwischen ihnen besteht. Ein Baum hat einen **Wurzelknoten** (root node), von dem aus alle anderen Knoten erreicht werden können. Jeder Knoten (außer der Wurzel) hat genau einen Elternknoten und kann 0 oder mehrere Kinderknoten haben.

Knoten
Kanten
Wurzelknoten

- (innerer) **Knoten**: Knoten mit Eltern und Kindern — Knoten
- **Kante**: Verbindungen zwischen den Knoten — Kante
- **Wurzel**: Knoten ohne Eltern — Wurzel
- **Blatt**: Knoten ohne Kinder — Blatt



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Methoden

- `insert`: fügt einen neuen Knoten in den Baum ein — insert
- `delete`: entfernt einen Knoten aus dem Baum — delete
- `search`: sucht nach einem bestimmten Knoten im Baum — search
- `traverse`: durchläuft alle Knoten im Baum in einer bestimmten Reihenfolge (z. B. Tiefensuche, Breitensuche) — traverse

Arten (Beispiele)

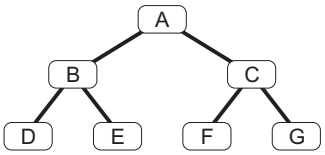
- *Binäre-Bäume*: zwei Kinder pro Knoten
- *AVL-Bäume*: Balancierter Binärbaum
- *B-Bäume*: Max und Min für die Anzahl der Kinder pro Knoten

Es gibt verschiedene Arten von Bäumen, wie z. B. binäre Bäume, AVL-Bäume, B-Bäume und Entscheidungsbäume. Jeder Typ hat seine eigenen Eigenschaften und Anwendungen.

- *Binäre Bäume* haben die Eigenschaft, dass jeder Knoten höchstens zwei Kinder hat. Sie werden häufig in Such- und Sortieralgorithmen verwendet.
- *AVL-Bäume* sind eine spezielle Art von binären Bäumen, die immer ausgeglichen sind und eine gute Balance zwischen Zeit- und Speicherkomplexität bieten.
- *B-Bäume* haben eine variable Anzahl von Kindern pro Knoten. Jeder Knoten (außer der Wurzel und der Blätter) hat mindestens eine bestimmte minimale Anzahl von Kindern und eine maximale Anzahl.

Ein Beispiel für die Verwendung eines Baumes wäre ein Suchalgorithmus, bei dem ein binärer Suchbaum verwendet wird, um schnell nach einem bestimmten Element in einer großen Menge an Daten zu suchen.

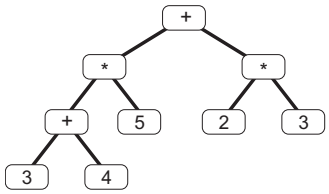
5.2 Binärbaum [45]



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

5.3 Beispiel: Arithmetischer Ausdruck [46]

Ausdruck `(3 + 4)* 5 + (2 * 3)` als Baum



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Arithmetische Ausdrücke können als Binärbäume dargestellt werden, weil sie eine Hierarchie von Operationen enthalten. In einem arithmetischen Ausdruck haben bestimmte Operationen Vorrang vor anderen, z. B. die Multiplikation vor der Addition. Ein Binärbaum kann diese Hierarchie effizient darstellen, indem er Operationen als Knoten und Operanden als Kinderknoten speichert. Die Operationen an den Knoten bestimmen die Art der Berechnung, die auf den Operanden durchgeführt werden soll, und die operanden werden rekursiv berechnet, bis man die endgültigen Werte erhält. Auf diese Weise kann man arithmetische Ausdrücke in einer natürlichen Weise als Baumstruktur darstellen und berechnen.

5.4 Binärbaum [47]

Binärbaums

Jeder Knoten eines **Binärbaums** kann durch eine einfache Datenstruktur abgebildet werden, die neben den Daten (`data`) die jeweils linken und rechten Kindknoten speichert (`left` und `right`).

Knoten eines Binärbaums

```
class Node {
    int data;
    Node left;
    Node right;

    public Node(int item) {
        key = item;
        left = right = null;
    }
}
```

Die Klasse für den Binärbaum enthält nur den Wurzelknoten und die Methoden, um auf dem Baum zu arbeiten.

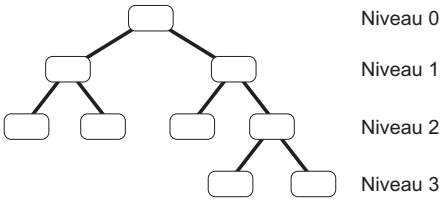
Baum

```
public class BinaryTree {
    Node root;
    ...
}
```

5.5 Binärbaum: Niveaus [48]

Niveaus

Ein Binärbaum kann in **Niveaus** unterteilt werden. Ein Niveau bezieht sich auf die Tiefe eines Knotens in dem Baum. Der Wurzelknoten befindet sich auf Niveau 0, die Kinder des Wurzelknotens auf Niveau 1, deren Kinder auf Niveau 2 und so weiter. Jeder Knoten auf einem höheren Niveau ist ein Vaterknoten für die Knoten auf einem niedrigeren Niveau. Die maximale Anzahl der Niveaus in einem Binärbaum ist durch die Tiefe des Baumes bestimmt, die die Länge des längsten Pfads von der Wurzel zu einem Blattknoten ist.



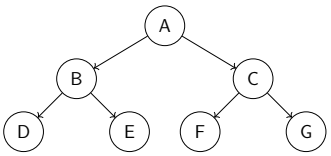
Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

5.6 Traversieren eines Baumes [49]

Es gibt drei Hauptwege, um durch die Knoten eines Binärbaumes zu durchlaufen: Tiefensuche (engl. Depth-First Search, DFS), Breitensuche (engl. Breadth-First Search, BFS) und die Symmetrische Reihenfolge (engl. In-Order traversal).

Durchlaufen von Knoten (**Traversierung**) — Traversierung

- **Tiefensuche** — Tiefensuche
  - **Pre-Order**: Knoten → linke Teilbaum → rechte Teilbaum — Pre-Order
  - **In-Order**: linke Teilbaum → Knoten → rechte Teilbaum — In-Order
  - **Post-Order**: linke Teilbaum → rechte Teilbaum → Knoten — Post-Order
- **Breitensuche**: alle Knoten auf demselben Niveau von links nach rechts besucht, dann wird auf das nächste Niveau gewechselt. Auch **Level-Order** genannt. — Breitensuche, Level-Order



Durchlauf	Reihenfolge
Pre-Order	A, B, D, E, C, F, G
In-Order	D, B, E, A, F, C, G
Post-Order	D, E, B, F, G, C, A
Breitensuche	A, B, C, D, E, F, G

- **Tiefensuche**: Die Tiefensuche durchläuft den Baum in einer rekursiven Art und Weise, indem zuerst der linke Teilbaum besucht wird, dann der Wurzelknoten und schließlich der rechte Teilbaum. Es gibt drei Arten von Tiefensuche, die man anwenden kann: — Tiefensuche
  - **Pre-Order**: Der Knoten wird besucht, bevor die Kinder besucht werden. — Pre-Order
  - **In-Order**: Der Knoten wird besucht, nachdem der linke Teilbaum besucht wurde und vor dem rechten Teilbaum. — In-Order
  - **Post-Order**: Der Knoten wird besucht, nachdem die Kinder besucht wurden. — Post-Order
- **Breitensuche**: Dieser Durchlauf besucht die Knoten eines Baumes in der Reihenfolge ihrer Tiefen. Es beginnt bei der Wurzel und besucht dann die Kinder der Wurzel in der Reihenfolge links, rechts und fährt dann fort, die Kinder der Kinder zu besuchen. — Breitensuche

Jeder dieser Durchläufe hat seine eigenen Anwendungen und Vorteile, je nach der Art der Aufgabe, die man lösen möchte. Beispielsweise ist die *Tiefensuche* (insbesondere die Post-Order-Traversierung) nützlich, wenn man die Freigabe von Speicher auf einem Baum durchführen möchte, da sie es ermöglicht, die Kinder zuerst freizugeben, bevor der Elternknoten freigegeben wird. Die *Breitensuche* ist nützlich, wenn man alle Knoten auf einer bestimmten Tiefe

besuchen möchte oder wenn man den kürzesten Weg zwischen zwei Knoten finden möchte. Die Symmetrische Reihenfolge ist nützlich, wenn man die Elemente in einem Baum sortiert zurückgeben möchte.

Die Tiefensuche lässt sich am einfachsten mit einem rekursiven Algorithmus realisieren.

Traversieren Pre-Order

```
public void preorder(Consumer<String> f, Node node) {
    if (node == null) return;
    f.accept(node.data); // Aktion auf dem Knoten ausführen
    preorder(f, node.left); // Linker Teilbaum
    preorder(f, node.right); // Rechter Teilbaum
}
```

Traversieren In-Order

```
public void inorder(Consumer<String> f, Node node) {
    if (node == null) return;
    inorder(f, node.left); // Linker Teilbaum
    f.accept(node.data); // Aktion auf dem Knoten ausführen
    inorder(f, node.right); // Rechter Teilbaum
}
```

Traversieren Post-Order

```
public void postorder(Consumer<String> f, Node node) {
    if (node == null) return;
    postorder(f, node.left); // Linker Teilbaum
    postorder(f, node.right); // Rechter Teilbaum
    f.accept(node.data); // Aktion auf dem Knoten ausführen
}
```

Bei der Breitensuche bietet sich ein iterativer Algorithmus an, der eine Queue als Hilfsdatenstruktur verwendet, um die Knoten auf derselben Ebene zu sammeln.

Traversieren Breitensuche

```
public void levelorder(Consumer<String> f) {
    var list = new LinkedList<Node>();
    list.add(root);
    while (!list.isEmpty()) {
        Node n = list.pollFirst();
        f.accept(n.data); // Aktion ausführen
        if (n.left != null) {
            list.add(n.left);
        }
        if (n.right != null) {
            list.add(n.right);
```

```
        }
    }
}
```

Kapitel 6

Graphen

6.1 Graphen [55]

Ein Graph ist eine abstrakte Datenstruktur, die **Knoten** (auch als Nodes bezeichnet) und **Kanten** (auch als Edges bezeichnet) enthält. Knoten repräsentieren die Objekte oder Entitäten innerhalb des Graphen, während Kanten die Beziehungen oder Verbindungen zwischen den Knoten darstellen.

Knoten

Kanten

Graph

Graph

- **Knoten** (nodes)
- **Kanten** (edges)

Knoten

Kanten

Arten

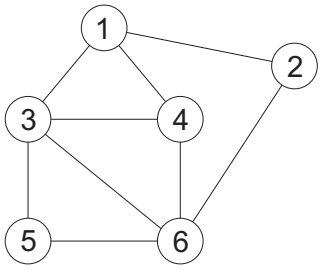
- Ungerichteter Graph
- Gerichteter Graph
- Gewichteter Graph

Graphen können ungerichtet sein, was bedeutet, dass die Kanten keine Richtung haben, oder gerichtet sein, was bedeutet, dass die Kanten eine Richtung haben. Graphen können auch gewichtet sein, was bedeutet, dass jede Kante ein Gewicht hat. Sie können verwendet werden, um verschiedene Arten von Daten und Beziehungen zwischen ihnen darzustellen, wie z. B. Straßenkarten, soziale Netzwerke und Abhängigkeiten in Computerprogrammen.

6.2 Ungerichteter Graph [56]

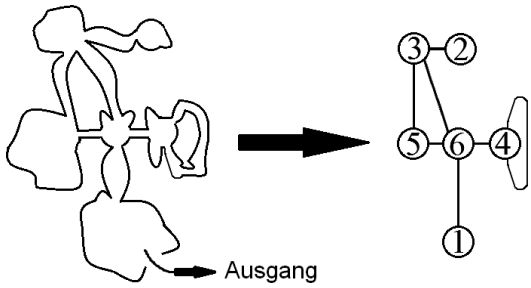
Ungerichteter Graph: Kanten geben keine Richtung vor

Ungerichteter Graph



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Höhlensystem als Beispiel für einen Graphen

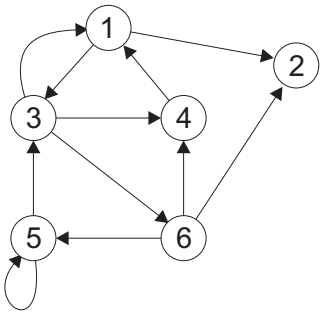


Quelle: René Krooß: Algorithmen und Datenstrukturen

6.3 Gerichteter Graph [58]

Gerichteter Graph: Kanten definieren eine Richtung

Gerichteter Graph

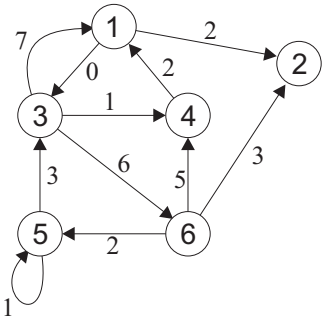


Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

6.4 Gewichteter Graph [59]

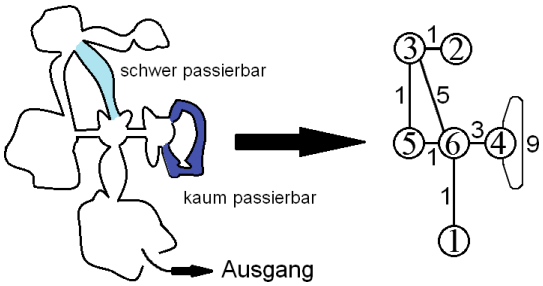
Gewichteter Graph: Kanten sind mit einem Gewicht annotiert

Gewichteter Graph



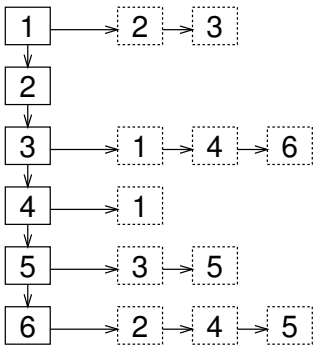
Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Höhlensystem als gewichteter Graph



Quelle: René Krooß: Algorithmen und Datenstrukturen

6.5 Graph als dynamische Datenstruktur [61]



Quelle: Gunter Saake und Kai-Uwe Sattler: Algorithmen und Datenstrukturen, 6. Auflage

Index

Abstrakte Datentypen, 2
abstrakter Datentyp, 2
ADT, 2

Binärbaums, 21
Blatt, 19
Breitensuche, 22
Buckets, 15

Cursor, 12

doppelt verketteten Liste, 10

einfach verkettete Liste, 7

Gerichteter Graph, 26
Gewichteter Graph, 26
Graph, 25

Hash-Tabelle, 14
head, 7

In-Order, 22
Iterator, 12

Kante, 19
Kanten, 19, 25
Kellerspeicher, 3
Knoten, 19, 25
Konkrete Datentypen, 2

Level-Order, 22
LIFO, 3
Liste, 2, 3

Niveaus, 21

Offenes Hashing, 17

Post-Order, 22
Pre-Order, 22
Priority Queue, 4

Ringpuffer, 5

Stack, 3, 4
Stapel, 3

tail, 7
Tiefensuche, 22
Traversierung, 22

Ungerichteter Graph, 25

verkettete Liste, 6

Wurzel, 19
Wurzelknoten, 19

Zeiger, 12

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Collections



Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule mannheim

Wintersemester 2025/2026

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**
- 1.1 Fragen [5] . . . 1
- 1.2 Begriffe [6] . . . 1
- 1.3 Links zur Java-Insel [7] . . . 1

2 Motivation und Grundlagen **2**
- 2.1 Was sind Collections? [9] . . . 2
- 2.2 Die Collection-Hierarchie [10] . . . 3

3 Iterable und Iterator **4**
- 3.1 Iterable und Iterator [13] . . . 4
- 3.2 Beispiel: Iterable und Iterator [14] . . . 4
- 3.3 ListIterator [16] . . . 6
- 3.4 Beispiel: ListIterator [17] . . . 6
- 3.5 Iterable und Iterator [18] . . . 7
- 3.6 Eigenes Iterable schreiben [19] . . . 7
- 3.7 Beispiel: Eigenes Iterable schreiben (Klasse) [20] . . . 8
- 3.8 Beispiel: Eigenes Iterable schreiben (innere Klasse) [22] . . . 9

4 Die Collection Interfaces **10**
- 4.1 Die Interfaces [24] . . . 10
- 4.2 Beispiel: Set [27] . . . 11
- 4.3 Beispiel: List [28] . . . 12
- 4.4 Methoden auf Collection [29] . . . 13
- 4.5 Beispiel: Collection [31] . . . 14
- 4.6 Methoden auf List [33] . . . 15
- 4.7 Beispiel: List [35] . . . 15
- 4.8 Methoden auf Set [37] . . . 16
- 4.9 Vorsicht: Veränderbare Objekte im Set [38] . . . 16
- 4.10 Methoden auf Queue [39] . . . 17
- 4.11 Das Deque [40] . . . 17

5 Maps **19**
- 5.1 Map und HashMap [42] . . . 19
- 5.2 Methoden von Map<K, V> [44] . . . 20

5.3 Methoden von Map<K, V> [45] . . . 20
5.4 Beispiel: Map und HashMap [46] . . . 20
5.5 Beispiel: Komplexer Schlüssel [48] . . . 21
5.6 Beispiel: Gleiche Schlüssel [49] . . . 22
5.7 Vorsicht: Veränderbare Objekte als Key [50] . . . 22

6 Collection Klassen 23
6.1 Klassen [52] . . . 23
6.2 Klassen und Implementierung [53] . . . 23
6.3 Die Sache mit null [56] . . . 24
6.4 Die Klasse Collections [57] . . . 25
6.5 Das alte Collections-API [59] . . . 25

7 Ordnen und Sortieren 26
7.1 Geordnet vs. sortiert [61] . . . 26
7.2 Comparable und Comparator [62] . . . 26
7.3 Beispiel: Comparable [63] . . . 27
7.4 Beispiel: Comparable (mit ArrayList) [65] . . . 28
7.5 Beispiel: Comparable (mit TreeSet) [66] . . . 29
7.6 Beispiel: Comparator [67] . . . 29
7.7 Beispiel: Comparator (mit ArrayList) [69] . . . 30
7.8 Beispiel: Comparator (mit TreeSet) [70] . . . 30
7.9 Methoden von SortedSet [71] . . . 31
7.10 Beispiel: SortedSet [72] . . . 31
7.11 Beispiel: Sortieren mit Collections.sort [73] . . . 32

8 Enumerationen und Collections 33
8.1 EnumSet und EnumMap [76] . . . 33
8.2 Beispiel: Todesstern mit EnumSet [77] . . . 33
8.3 Beispiel: Hilfsmethoden auf EnumSet [78] . . . 34
8.4 Beispiel: Ohne EnumSet (old school) [79] . . . 35
8.5 Beispiel: Mit EnumSet [82] . . . 36

9 Collections und Nebenläufigkeit 37
9.1 ConcurrentModificationException [86] . . . 37
9.2 Beispiel: ConcurrentModificationException [87] . . . 37
9.3 Thread-Safety [88] . . . 38
9.4 ConcurrentHashMap<K, V> [89] . . . 38
9.5 ConcurrentHashMap<K, V> [90] . . . 39
9.6 Beispiel: ConcurrentHashMap [91] . . . 39
9.7 CopyOnWriteArrayList<E> [92] . . . 39

Index ii

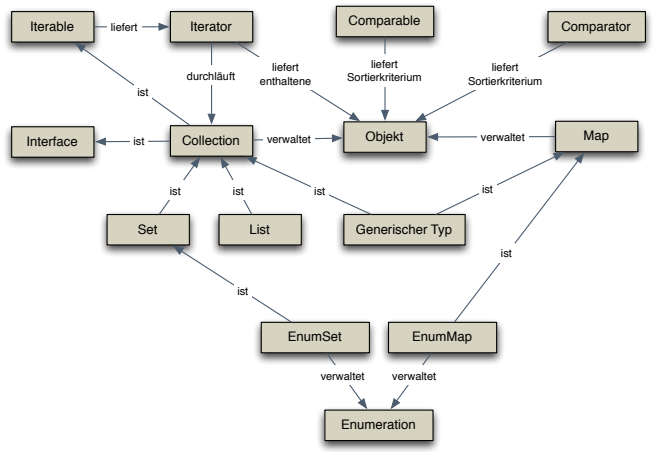
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie kann ich die Beschränkungen der Java-Arrays abschütteln?
- Wie kann ich Mengen von Objekten sinnvoll verwalten?
- Wie kann ich Daten sortieren und schnell wiederfinden?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 18.1 Listen
- 18.2 Mengen (Sets)
- 4.6 Assoziative Speicher

Kapitel 2

Motivation und Grundlagen

2.1 Was sind Collections? [9]

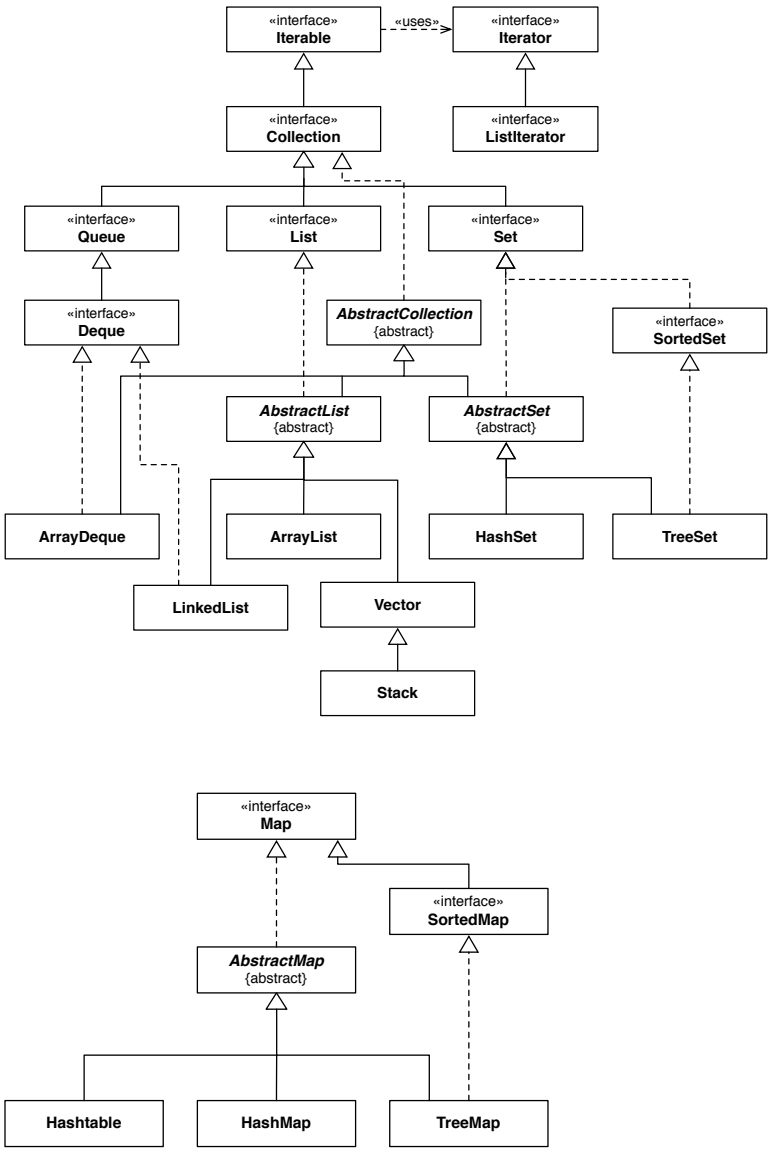
Collections

Collections …

- … dienen der Verwaltung von Mengen von Objekten („Collection“ = „Sammlung“)
- … verwalten eine Reihe von Objekten (die Elemente der Collection) als ein Objekt
- … sind im Gegensatz zu Arrays dynamisch
- … besitzen viele hilfreiche Methoden
- … stellen einen der nützlichsten Teil der Klassenbibliothek dar

Bisher ist die einzige Methode, die wir kennen, um Objekte zu sammeln und Mengen von Objekten zu verwalten das Array. Da Arrays in Java aber nicht dynamisch sind, d. h. bei der Erzeugung einmal eine feste Größe bekommen, die danach nicht mehr geändert werden kann, ist die Verwendung von Arrays relativ umständlich und fehleranfällig. Aus diesem Grund gibt es in der Java-Klassenbibliothek eine ganze Reihe von Klassen, die als sehr nützlicher Ersatz von Arrays genutzt werden können. Die Collections sind so nützlich, dass in der professionellen Java-Entwicklung Arrays nur selten zum Einsatz kommen, wenn sie besondere Vorteile gegenüber Collections haben, z. B. eine höhere Geschwindigkeit.

2.2 Die Collection-Hierarchie [10]



Das Diagramm zeigt einen Überblick über alle Interfaces und Klassen aus dem sogenannten **Collection-API**. Die Zusammenhänge und Aufgaben der einzelnen Klassen werden wir im Folgenden schrittweise beleuchten.

Collection-API

Wie man sieht, gibt es zwei unterschiedliche Hierarchien: die von `Collection` abgeleiteten Interfaces (und Klassen) und die von `Map` abgeleiteten.

Kapitel 3

Iterable und Iterator

3.1 Iterable und Iterator [13]

- `Iterable`-Instanzen liefern einen Iterator zurück — `Iterable`
- `Iterator` dient dem Durchlaufen von Collections (iterieren) — `Iterator`
- Iterable und Iterator sind die Grundlage für die foreach-Schleife
- Methoden auf `Iterator<T>`
  - `boolean hasNext()` - Gibt an, ob noch Elemente vorhanden sind — `hasNext()`
  - `T next()` - Geht zum nächsten Element und gibt es zurück — `next()`
  - `remove()` - Entfernt das aktuelle Element (muss nicht unterstützt werden) — `remove()`
- Iterator kann nur vorwärts laufen, für bestimmte Collections gibt es den `ListIterator`, der rückwärts navigieren kann

Immer wenn man mit Sammlungen von Objekten hantiert ist das Durchlaufen der Sammlung (**iterieren**) eine der grundlegenden Operationen. Man möchte sukzessive die Objekte aus der Sammlung anschauen. Hierzu dienen die Interfaces `Iterable` und `Iterator`. Etwas verwirrend mag erscheinen, dass man die Collection offensichtlich nur vorwärts durchlaufen kann und nicht wahlfrei auf einzelne Elemente zugreifen kann – man kann nicht sagen „gibt mir das 5. Element“. Der Grund wird später zwar noch genauer beleuchtet, hier sei aber schon einmal angemerkt, dass es Collections gibt (`Set`), in denen die Objekte ohne jede Ordnung herumliegen und damit gar kein 5. Element existiert. — `iterieren`

Wenn der Iterator bereits beim letzten Element der Collection angekommen ist, wirft `next()` eine Ausnahme vom Typ `NoSuchElementException`.

`remove()` muss von Collections nicht unterstützt werden. Wenn eine Collection diese Methode nicht unterstützt, darf der Iterator eine `UnsupportedOperationException` werfen.

3.2 Beispiel: Iterable und Iterator [14]

```
ArrayList<String> list = new ArrayList<>();
list.add("Eintrag 1");
list.add("Eintrag 2");
```

```
list.add("Eintrag 3");

Iterable<String> iterable = list;

Iterator<String> iterator = iterable.iterator();

while (iterator.hasNext()) {
    String wert = iterator.next();
    System.out.println(wert);
}
```

```
Eintrag 1
Eintrag 2
Eintrag 3
```



Dieses Beispiel zeigt eine explizite Benutzung des Iterators mit einer `while`-Schleife. Die Klasse `ArrayList` werden wir später noch kennenlernen. Hier sei nur angemerkt, dass es sich um eine der Collection-Klassen handelt und mit `add` Elemente hinzugefügt werden können.

Nachdem der Iterator angelegt wurde, läuft die `while`-Schleife so lange durch die Elemente, bis kein weiteres mehr vorhanden ist (`iterator.hasNext()` liefert `false`). Das jeweils aktuelle Element wird von `iterator.next()` zurückgeliefert.

Die Zuweisung `Iterable<String> iterable = list;` dient nur dazu klarzumachen, dass `ArrayList` das `Iterable`-Interface implementiert. Normalerweise würde man sich diese sparen und eher schreiben:

```
ArrayList<String> list = new ArrayList<>();
list.add("Eintrag 1");
list.add("Eintrag 2");
list.add("Eintrag 3");

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {
    String wert = iterator.next();
    System.out.println(wert);
}
```



```
ArrayList<String> list = new ArrayList<>();
list.add("Eintrag 1");
list.add("Eintrag 2");
list.add("Eintrag 3");

for (String wert : list) {
    System.out.println(wert);
```



```
}
```

```
Eintrag 1
Eintrag 2
Eintrag 3
```



foreach

Da die **foreach**-Schleife nicht nur mit Arrays, sondern mit allen Objekten umgehen kann, deren Klasse das Interface `Iterable` implementiert, kann man das Beispiel durch eine solche Schleife noch einmal verkürzen.

Dieses Beispiel ist semantisch äquivalent mit dem vorhergehenden. Es ist sogar so, dass der Java-Compiler die hier dargestellte foreach-Schleife in die vorher dargestellte `while`-Schleife umsetzt. Trotzdem ist die foreach-Variante übersichtlicher und daher vorzuziehen.

3.3 ListIterator [16]

ListIterator

`ListIterator` erweitert `Iterator` um zusätzliche Methoden

- `boolean hasPrevious()` – Gibt an, ob es noch vorhergehende Elemente gibt
- `T previous()` – Geht zum vorhergehenden Element und gibt es zurück
- `int nextIndex()` – Gibt die Position des nächsten Elements an
- `int previousIndex()` - Gibt die Position des vorhergehenden Elements an
- `set(T element)` – Ersetzt das aktuelle Element
- `add(T element)` – Fügt hinter dem aktuellen Element ein weiteres ein

Ist nur auf bestimmten Collection-Arten möglich

Wenn die Collection in der Lage ist, den Objekten einen Index zuzuordnen, kann anstatt des `Iterator` der `ListIterator` verwendet werden. Dieser leitet von `Iterator` ab und fügt weitere Methoden hinzu.

Die `add`-Operation ist optional und muss vom konkreten Iterator nicht unterstützt werden.

3.4 Beispiel: ListIterator [17]

```
ArrayList<String> list = new ArrayList<>();
list.add("Eintrag 1"); list.add("Eintrag 2"); list.add("Eintrag 3");

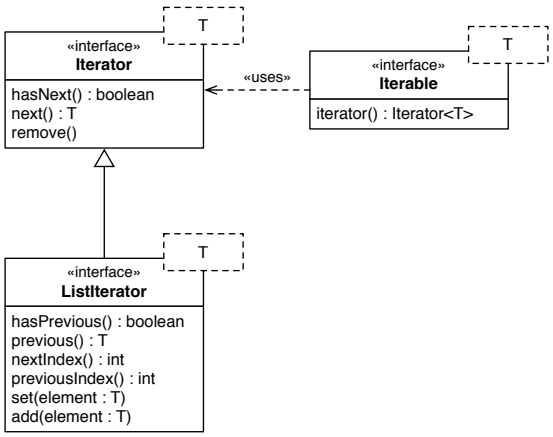
ListIterator<String> iterator = list.listIterator();
iterator.next(); iterator.next();
iterator.previous();
iterator.set("Neuer Eintrag 2");
iterator.next(); iterator.next();
iterator.add("Neuer Eintrag 4");
```



```
for (String wert : list) {
    System.out.println(wert);
}
```

```
Eintrag 1
Neuer Eintrag 2
Eintrag 3
Neuer Eintrag 4
```

3.5 Iterable und Iterator [18]



Die Grafik zeigt noch einmal den Zusammenhang zwischen `Iterable`, `Iterator` und `ListIterator`. Wenn eine Klasse iterierbar sein soll, d. h. wenn man Elemente über einen `Iterator` aus der Klasse auslesen kann, muss sie das Interface `Iterable` implementieren. Wenn sie das Interface implementiert, so hat sie eine Methode `iterator()`, über die man einen `Iterator` anfordern kann. Eine erweiterte Form des Iterators wird durch den `ListIterator` zur Verfügung gestellt.

Die Trennung zwischen `Iterable` und `Iterator` ist wichtig, damit man zu einer Klasse mehrere Iteratoren erzeugen kann. Man kann also mehrfach `iterator()` aufrufen und dann mit den zurückgegebenen Objekten gleichzeitig über die Elemente der Collection laufen. Hierbei merkt sich jeder Iterator, an welcher Stelle er sich gerade befindet.

3.6 Eigenes Iterable schreiben [19]

- Man kann eigene Klassen in `foreach`-Schleifen verwenden
- Hierzu implementiert die Klasse das Interface `Iterable` und stellt eine eigene Implementierung von `Iterator` zur Verfügung
  - als separate Klasse
  - als innere Klasse

Es ist nicht ungewöhnlich, mit einer selbstgeschriebenen Klasse das Interface `Iterable` zu implementieren, um durch die Elemente per foreach-Schleife laufen zu können. Hierzu muss man vor allem den Iterator schreiben, der von der `iterator()`-Methode zurückgegeben wird.

Für die Implementierung des Iterators stehen zwei Möglichkeiten zur Wahl: entweder man schreibt ihn als separate Klasse oder man greift, was deutlich einfacher ist, auf eine innere Klasse zurück.

3.7 Beispiel: Eigenes Iterable schreiben (Klasse) [20]

```
public class IterableMitKlasse implements Iterable<String> {
    private String[] stringListe;
    ...
    public Iterator<String> iterator() {
        return new MyIterator(stringListe);
    }
}
```

```
class MyIterator implements Iterator<String> {
    private int pos = 0;
    private String[] stringListe;

    public MyIterator(String[] stringListe) {
        this.stringListe = stringListe;
    }

    public boolean hasNext() {
        return (pos < stringListe.length);
    }

    public String next() {
        return stringListe[pos++];
    }

    public void remove() { throw new UnsupportedOperationException(); }
}
```

Wählt man den Weg über eine getrennte Klasse, muss dem Iterator über den Konstruktor eine Referenz auf die Elemente übergeben werden (hier `stringListe`). Die Position in der Liste muss der Iterator selbst verwalten, damit es möglich ist, mehrere Iteratoren zu haben, die über dasselbe Objekt laufen.

3.8 Beispiel: Eigenes Iterable schreiben (innere Klasse) [22]

```
public class IterableInnereKlasse implements Iterable<String> {
    private String[] stringListe;

    public Iterator<String> iterator() {
        return new Iterator<String>() {
            private int pos = 0;

            public boolean hasNext() {
                return (pos < stringListe.length);
            }
            public String next() {
                return stringListe[pos++];
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }};
    }
}
```

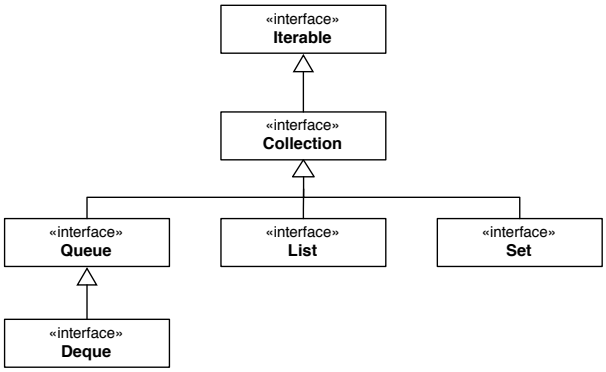


Bei einer inneren Klasse (hier einer anonymen) ist der Zugriff auf die Daten des umgebenden Objektes einfacher, sodass der Iterator mit weniger Aufwand zu implementieren ist. Die Position muss er aber auch in diesem Fall selbst verwalten.

Kapitel 4

Die Collection Interfaces

4.1 Die Interfaces [24]



Das Collection-API zerfällt in fünf grundlegende Interfaces und deren Implementierungen. Da die Verwendung und die wichtigsten Eigenschaften durch die Interfaces festgelegt werden, betrachten wir zuerst die Interfaces und dann in einem zweiten Schritt die dazugehörigen Klassen.

- `Collection` — `Collection`
  - Sammlung von Objekten ohne besondere Ordnung
  - Duplikate sind zulässig
- `List` — `List`
  - Geordnete Sammlung von Objekten
  - Duplikate sind zulässig
- `Set` — `Set`
  - Ungeordnete Sammlung von Objekten
  - Duplikate sind nicht zulässig
- `Queue` — `Queue`
  - geordnet, Duplikate erlaubt

    - Warteschlange mit speziellen Methoden, die mit leeren oder überfüllten Warteschlangen ohne Ausnahmen umgehen können

- `Deque` Deque
    - geordnet, Duplikate erlaubt
    - Erweiterte Form der `Queue` mit Operationen an beiden Enden der Warteschlange

Die Basis aller Interfaces ist das Interface `Collection`, in dem die Methoden festgelegt werden, die allen Collections gemeinsam sind. Generell sind zwei Fragen bei allen Interfaces wichtig:

1. Wird eine **Ordnung** gehalten? Ordnung bedeutet in diesem Zusammenhang, dass die Elemente innerhalb der Collection in einer festen Reihenfolge abgelegt sind, die sich aus der Reihenfolge des Einfügens ergibt. Wenn eine Ordnung vorhanden ist, kann man den gespeicherten Objekten stabile **Indices** zuordnen, jedes Objekt bekommt einen Index, der über die Zeit stabil bleibt. Hält die Collection keine Ordnung, können die Elemente nicht indiziert werden und bei aufeinanderfolgenden Durchläufen mit einem Iterator können die Objekte in unterschiedlicher Sequenz ausgegeben werden. Insbesondere kommen bei einer ungeordneten Collection die Elemente in einer anderen Reihenfolge heraus, als man sie hineingesteckt hat. Ordnung Indices
2. Sind **Duplikate** zulässig? Duplikate sind Elemente gleichen Inhalts. Wenn Duplikate zulässig sind, können in einer Collection mehrere Objekte enthalten sein, bei denen ein `equals()`-Aufruf `true` ergeben würde. Sind Duplikate verboten, gibt es keine zwei Objekte in der Collection bei denen `equals()` `true` ergeben würde. Duplikate

`Collection` trifft zu den beiden Fragen keine abschließende Aussage, d. h. es gibt keine Ordnung und Duplikate sind zulässig. Das genaue Verhalten wird erst auf der Ebene der Subinterfaces festgelegt.

`List` erlaubt Duplikate und hält eine Ordnung. Im Gegensatz dazu verbietet `Set` Duplikate und garantiert keine Ordnung. `Queue` stell eine Warteschlange dar und `Deque` erweitert diese um Operationen an beiden Enden, sodass man sowohl eine Queue als auch einen Stack damit realisieren kann. Sowohl `Queue` als auch `Deque` sind geordnet und erlauben Duplikate.

4.2 Beispiel: Set [27]

```
Set<String> set = new HashSet<>();

set.add("Element A");
set.add("Element B");
set.add("Element C");
set.add("Element A");
set.add("Element B");

for (String element : set) {
    System.out.println(element);
}
```

```
Element B
Element C
Element A
```

Im Beispiel kann man sehen, dass bei einem `Set` die Elemente keine Ordnung halten und dass die beiden Duplikate „Element A“ und „Element B“ nur jeweils einmal aufgenommen wurden.

Ein wichtiger Punkt ist, dass durch das Hinzufügen eines Objektes zu einer Collection (hier einem `Set`) keine Kopie des Objektes angelegt wird. Die Collection speichert nur eine Kopie der *Referenz*, d. h. der Inhalt des Objektes kann über eine weitere Referenz, so noch vorhanden, verändert werden. Das folgende Beispiel zeigt diesen Sachverhalt:

```
public class A {
    public int wert;
    public String toString() {
        return "" + wert;
    }
}
```

```
A a = new A();
a.wert = 42;

Set<A> set = new HashSet<>();
set.add(a);
System.out.println(set); // -> [42]

a.wert = 23;
System.out.println(set); // -> [23]
```

4.3 Beispiel: List [28]

```
List<String> list = new ArrayList<>();

list.add("Element A");
list.add("Element B");
list.add("Element C");
list.add("Element A");
list.add("Element B");

for (String element : list) {
    System.out.println(element);
}
```

```
Element A
Element B
Element C
Element A
Element B
```

Die Liste hält im Gegensatz zum Set die Ordnung und erlaubt Duplikate. Deswegen gibt es „Element A“ und „Element B“ dann zweimal.

4.4 Methoden auf Collection [29]

- `boolean add(E e)` – Fügt ein Element `e` hinzu
- `boolean addAll(Collection<? extends E> c)` – Fügt alle Elemente aus `c` hinzu
- `boolean remove(Object o)` – Entfernt das gegebene Objekt `o` falls vorhanden
- `boolean removeAll(Collection<?> c)` – Entfernt alle Elemente, die in der gegebenen Collection `c` enthalten sind
- `boolean retainAll(Collection<?> c)` – Entfernt alle Elemente, die nicht in der gegebenen Collection `c` enthalten sind
- `void clear()` – Entfernt alle Elemente
- `boolean contains(Object o)` – Prüft, ob das gegebene Objekt in der Collection enthalten ist
- `boolean containsAll(Collection<?> c)` – Prüft ob alle Elemente der gegebenen Collection enthalten sind
- `int size()` – Gibt Anzahl der Elemente zurück
- `boolean isEmpty()` – `(size()== 0)`
- `Object[] toArray()` – Konvertiert die Collection in ein Object-Array
- `T[] toArray(T[] a)` – Konvertiert die Collection in ein Array

Die Methoden aus dem Interface `Collection` sind weitgehend selbsterklärend. Die Konvertierung in ein Array ist allerdings etwas ungewöhnlich, da es hier zwei Varianten gibt:

- `Object[] toArray()` – Die Elemente der Collection werden in ein Array vom Typ `Object[]` konvertiert. Das Array hat dabei so viele Elemente, wie die Collection (`size()`). Da man bei einem Object-Array jedes Element einzeln casten muss, wenn man damit etwas anfangen möchte, kommt diese Methode selten zum Einsatz.
- `T[] toArray(T[] a)` – Konvertiert die Collection in ein Array vom Typ, der durch den Typ-Parameter `T` der Collection festgelegt wurde. Bei diese Methode muss man ein Array vom richtigen Typ übergeben. Ist dieses Array *groß genug* für alle Elemente der Collection, so werden diese einfach in das Array umkopiert und eine Referenz auf das übergebene Array wird zurückgegeben. Ist das übergebene Array zu klein, wird ein

neues, hinreichend großes Array angelegt und mit den Elementen befüllt. Zurückgegeben wird dann eine Referenz auf das neu erzeugte Array.

Üblicherweise ruft man die `toArray`-Methode mit folgender Konstruktion auf (angenommen die Collection sei vom Typ `Collection<String>`:

```
String[] array = collection.toArray(new String[collection.size()]);
```

4.5 Beispiel: Collection [31]

```
Collection<String> col1 = new ArrayList<>();

col1.add("Element 1");
col1.add("Element 2");
col1.add("Element 3");
col1.add("Element 4");

System.out.println(col1.size()); // --> 4
System.out.println(col1.contains("Element 3")); // --> true

col1.remove("Element 3");
System.out.println(col1.size()); // --> 3

Collection<String> col2 = new ArrayList<>();
col2.addAll(col1);

System.out.println(col2); // --> [Element 1, Element 2, Element 4]
```

```
col1.clear();
System.out.println(col1.size()); // --> 0

col1.add("Element 2");
col2.removeAll(col1);

System.out.println(col2); // --> [Element 1, Element 4]

String[] str1 = col1.toArray(new String[0]);

Object[] str2 = col2.toArray();

System.out.println(Arrays.asList(str1)); // --> [Element 2]
System.out.println(Arrays.asList(str2)); // --> [Element 1, Element 4]
```

4.6 Methoden auf List [33]

Zusätzlich zu den Methoden aus Collection hat **List** noch

List

- `boolean addAll(int index, Collection<? extends E> c)` – Fügt alle Elemente von `c` an der gegebenen Stelle ein und verschiebt die existierenden Elemente
- `E get(int index)` – Gibt das Element an der gegebenen Position zurück
- `E set(int index, E element)` – Ersetze das Element an der gegebenen Position
- `void add(int index, E element)` – Füge an der gegebenen Position ein Element ein
- `E remove(int index)` – Entferne das Element an der Position
- `int indexOf(Object o)` – Position des gegebenen Objekts (erster Treffer)
- `int lastIndexOf(Object o)` – Position des gegebenen Objekts (letzter Treffer)
- `ListIterator<E> listIterator()` – Liefert einen ListIterator
- `ListIterator<E> listIterator(int index)` – Liefert einen ListIterator ab der gegebenen Position
- `List<E> subList(int fromIndex, int toIndex)` – Erzeugt eine Sub-Liste beginnend bei `fromIndex` (inklusive) und endend bei `toIndex` (exklusive)

`List` erweitert die Methoden von `Collection` um Methoden, die ausnutzen, dass in einer `List` alle Elemente einen eindeutigen Index haben.

4.7 Beispiel: List [35]

```
List<String> list1 = new ArrayList<String>();

list1.add("Element 1"); list1.add("Element 2");
list1.add("Element 3"); list1.add("Element 4");

System.out.println(list1.get(3)); // --> Element 4
list1.remove(0);
System.out.println(list1); // --> [Element 2, Element 3, Element 4]

list1.add(2, "Eingefuegt");
System.out.println(list1); // --> [Element 2, Element 3, Eingefuegt,
                           // Element 4]

List<String> list2 = list1.subList(1, 3);
System.out.println(list2); // --> [Element 3, Eigefuegt]
```

```
list2.set(1, "Ausgetauscht");
System.out.println(list2); // --> [Element 3, Ausgetauscht]

list2.add(2, "Eingefuegt");
System.out.println(list2); // --> [Element 3, Ausgetauscht, Eingefuegt]
```

```
System.out.println(list2.indexOf("Element 3")); // --> 0
```

4.8 Methoden auf Set [37]

`Set` fügt zur Schnittstelle von Collection keine weiteren Methoden hinzu, sondern definiert nur die Semantik

Set

- ungeordnet
- keine Duplikate zulässig

`Set` hat keine eigenen Methoden, weil Collection schon alles definiert hat, was möglich ist.

4.9 Vorsicht: Veränderbare Objekte im Set [38]

- Das Verhalten hash-basierter Sets ist undefiniert, wenn der Wert nach dem Hinzufügen verändert wird
- Schlimmstenfalls werden die Objekte nie mehr wiedergefunden
- Daher sollte man vorsichtig bei veränderbaren Objekte (mutuable objects) in Sets sein
- Eine Set darf sich *nicht* selbst enthalten

Sets werden normalerweise mithilfe von *Hash-Tabellen* implementiert. Diese sind aber empfindlich gegen Veränderungen des Inhalts von Objekten, nachdem diese einmal hinzugefügt wurden. Deshalb darf man Objekte nach dem Speichern in einem `Set` nicht mehr verändern. Will man sie doch manipulieren muss man

1. das Objekt aus dem Set entfernen (`remove()`)
2. das Objekt verändern
3. das Objekt dem Set wieder hinzufügen (`add()`)

Der Grund liegt darin, dass bei einer Hash-Tabelle die Speicherposition anhand des Hash-Wertes (`hashCode()`) bestimmt wird. Der Hash-Wert berechnet sich aus dem Inhalt des Objekts. Ändert sich der Inhalt, so ändert sich der Hash-Wert. Verändert man den Inhalt und damit den Hash-Wert, so ist das Objekt in der Hash-Tabelle noch immer an der Stelle gespeichert, die seinem alten Hash-Wert entspricht – die Position wird nur einmalig beim Einfügen bestimmt. Damit entspricht die Speicherposition nicht mehr dem aktuellen Hash-Wert und das Objekt kann in der Tabelle nicht wiedergefunden werden.

Verändert man Objekte nachträglich kommt es zu der paradoxen Situation, dass die `contains()`-Methode meldet, dass das Objekt nicht vorhanden ist, beim Iterieren wird es aber ausgegeben.

4.10 Methoden auf Queue [39]

Zusätzlich zu den Methoden aus Collection hat `Queue` noch

`Queue`

- `boolean offer(E e)` – Versucht ein Element am Ende der Warteschlange anzufügen. Wirft keine Ausnahme wenn das nicht klappt, sondern gibt dann einfach `false` zurück

  `offer(E e)`
- `E poll()` – Holt das erste Element der Warteschlange und entfernt es. Wenn die Queue leer ist, gibt die Methode `null` zurück

  `poll()`
- `E peek()` – Holt das erste Element der Warteschlange, entfernt es aber nicht. Wenn die Queue leer ist, gibt die Methode `null` zurück

  `peek()`
- `E element()` – Holt das erste Element der Warteschlange, entfernt es aber nicht. Wenn die Queue leer ist, wirft die Methode eine `NoSuchElementException`

  `element()`

Mit dem Interface `Queue` wird eine Warteschlange (Ringpuffer) zur Verfügung gestellt.

4.11 Das Deque [40]

`Deque` (sprich „deck“) steht für double ended queue

`Deque`

Man kann Elemente am Anfang und am Ende hinzufügen und entfernen

- `void addFirst(E e)`, `void addLast(E e)`
- `Iterator<E> descendingIterator()`
- `E getFirst()`, `E getLast()`
- `boolean offerFirst(E e)`, `boolean offerLast(E e)`
- `E peekFirst()`, `E peekLast()`
- `E pollFirst()`, `E pollLast()`
- `E removeFirst()`, `E removeLast()`

Man kann ein `Deque` sowohl als Queue als auch als Stack verwenden. Insofern ist `java.util.Stack` überflüssig und kann durch ein Deque ersetzt werden.

Als Queue

- Queue-Methode → Deque-Methode
- `add(e)` → `addLast(e)`
- `offer(e)` → `offerLast(e)`
- `remove()` → `removeFirst()`
- `poll()` → `pollFirst()`
- `element()` → `getFirst()`
- `peek()` → `peekFirst()`

Als Stack

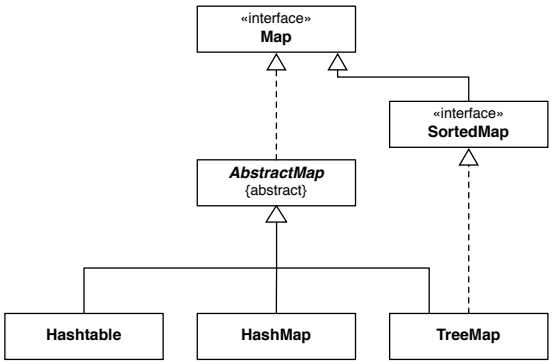
- Stack-Methode → Deque-Methode
- `push(e)` → `addFirst(e)`

- `pop()` → `removeFirst()`
- `peek()` → `peekFirst()`

Kapitel 5

Maps

5.1 Map und HashMap [42]



- Maps dienen dem Erstellen von **Wörterbüchern** (dictionaries)
- Einem **Schlüssel** (key) wird ein **Wert** (value) zugeordnet
- Die gängigste Implementierung des `Map`-Interfaces ist `HashMap`, die sich auf dem Hashwerten (`hashCode()`) der Objekte abstützt
- Bei gut verteilten Hashwerten ist das Lesen eines Objektes aus einer HashMap O(1)

Wörterbüchern

Schlüssel

Wert

Während die Aufgabe der Collections darin besteht, Mengen von Objekte zu speichern, geht es bei einer `Map` um den Aufbau einer Datenstruktur aus **Schlüssel-Wert-Paaren** (key/value pairs). Hierbei werden Objekte als Schlüssel für andere Objekte verwendet. Der klassische Anwendungsfall hierfür ist ein Cache, z. B. im Browser: der Schlüssel ist die URL (`String`), der Wert der Inhalt der zwischengespeicherten Webseite. Andere Namen für eine Map sind **assoziatives Array** oder einfach nur **Hash**.

Map

Schlüssel-Wert-Paaren

assoziatives Array

Hash

Da die am häufigsten verwendete Implementierung des `Map`-Interfaces auf Hash-Tabellen basiert, kann der Zugriff auf ein Objekt in **konstanter Zeit** erfolgen, d. h. die Menge der gespeicherten Objekte hat keine Auswirkung auf die Zeit, die das Finden und Auslesen braucht.

konstanter Zeit

Viele Programmiersprachen kennen das Konzept der *Map*, viele sogar als eingebautes Sprachmittel. So profitieren JavaScript, PHP und Ruby stark von den assoziativen Arrays.

5.2 Methoden von Map<K, V> [44]

- `int size()` – Anzahl der Elemente
- `boolean isEmpty()` – (`size() == 0`)
- `boolean containsKey(Object key)` – Testet, ob Schlüssel `key` vorhanden ist
- `boolean containsValue(Object value)` – Testet, ob Wert `value` vorhanden ist
- `V get(Object key)` – Liest den Wert zum Schlüssel `key`
- `V put(K key, V value)` – Setzt Schlüssel `key` und Wert `value`
- `V remove(Object key)` – Entfernt den Wert zu `key` (und `key` selbst)

5.3 Methoden von Map<K, V> [45]

- `void putAll(Map<? extends K, ? extends V> m)` – Fügt eine ganze Map hinzu
- `void clear()` – Löscht die Map
- `Set<K> keySet()` – Alle Schlüssel als Set
- `Collection<V> values()` – Alle Werte als Collection
- `Set<Map.Entry<K, V>> entrySet()` – Alle Einträge als spezielles Set

Viele Methoden aus dem Interface `Map` sind denen aus den Collections nicht unähnlich nur, dass hier entweder nach Schlüssel und Wert getrennt wird oder ein Schlüssel übergeben werden muss, um auf das Objekt zuzugreifen.

Man kann aus der Map über die Methoden `values` und `keySet` die Menge aller Werte bzw. aller Schlüssel auslesen. Üblicherweise verwendet man allerdings zum Iterieren über alle Elemente der Map die spezielle Methode `entrySet`, die in einem der folgenden Beispiele noch dargestellt wird.

5.4 Beispiel: Map und HashMap [46]

```
Map<String, String> map = new HashMap<>();

map.put("go", "gehen");
map.put("walk", "gehen");
map.put("sleep", "schlafen");
map.put("learn", "lernen");

System.out.println(map); // --> {learn=lernen, sleep=schlafen,
                         // walk=gehen, go=gehen}

System.out.println(map.get("go")); // --> gehen
System.out.println(map.get("walk")); // --> gehen
System.out.println(map.get("learn")); // --> lernen

System.out.println(map.keySet()); // --> [learn, sleep, walk, go]
System.out.println(map.values()); // --> [lernen, schlafen, gehen, gehen]
```

Das Beispiel zeigt die elementaren Operationen auf einer Map:

- Einfügen von Schlüssel/Wert-Paaren mit `put()`
- Auslesen von Elementen über den Schlüssel mit `get()`
- Auslesen aller Schlüssel mit `keySet()`
- Auslesen aller Werte mit `values()`

Zusätzlich sieht man, dass die `toString()`-Methode so überschrieben ist, dass der Inhalt der Map im Format `key=value` ausgegeben wird.

```
Set<Map.Entry<String, String>> entrySet = map.entrySet();

for (Map.Entry<String, String> entry : entrySet) {
    String key = entry.getKey();
    String value = entry.getValue();
    System.out.printf("%s engl. fuer %s%n", key, value);
}
```

Das Iterieren über den gesamten Inhalt einer Map erfolgt üblicherweise wie in diesem Beispiel dargestellt. Man kann über die Methode `entrySet()` ein `Set` erhalten, das aus speziellen Objekten vom Typ `Map.Entry` besteht. Jedes dieser Objekte enthält einen Schlüssel und den dazugehörigen Wert. Den Schlüssel kann man über `getKey()`, den Wert über `getValue()` aus dem `Map.Entry`-Objekt auslesen. Da `Map` ein generischer Typ mit den Parametern `K` für den Typ des Schlüssels und `V` für den Typ des Wertes ist, hat `Map.Entry` ebenfalls diese Typ-Parameter.

5.5 Beispiel: Komplexer Schlüssel [48]

```
Map<Personalnummer, Mitarbeiter> map = new HashMap<>();

Mitarbeiter meier = new Mitarbeiter("Meier", 1200.00);
Mitarbeiter mueller = new Mitarbeiter("Mueller", 2400.00);
Mitarbeiter schulze = new Mitarbeiter("Schulze", 1200.00);

map.put(new Personalnummer("4711"), meier);
map.put(new Personalnummer("4242"), mueller);
map.put(new Personalnummer("0007"), schulze);

System.out.println(map.containsKey(new Personalnummer("0007"))); // --> true
System.out.println(map.containsValue(meier)); // --> true
```

Obwohl sehr häufig `String` als Schlüssel für eine Map verwendet wird, kann jedes Objekt, das korrekt `equals()` und `hashCode()` überschreibt, als Schlüssel verwendet werden. Diese

Umstand demonstriert dieses Beispiel bei dem als Schlüssel eine Klasse `Personalnummer` verwendet wird.

5.6 Beispiel: Gleiche Schlüssel [49]

```
Map<String, String> map = new HashMap<>();

map.put("gehen", "go");
map.put("gehen", "walk");
map.put("schlafen", "sleep");
map.put("lernen", "learn");

System.out.println(map);
```



```
{gehen=walk, lernen=learn, schlafen=sleep}
```



Verwendet man in einer Map denselben Schlüssel mehrfach (hier `"gehen"`) so ersetzt der zweite Wert den ersten, d. h. im Beispiel wird nur `"walk"` aber nicht `"go"` gespeichert.

Aufgrund dieses Verhaltens ersetzt man bei einer Map einen Wert einfach, indem man einen anderen Wert mit demselben Schlüssel einfügt. Ein vorheriges Löschen ist nicht notwendig.

5.7 Vorsicht: Veränderbare Objekte als Key [50]

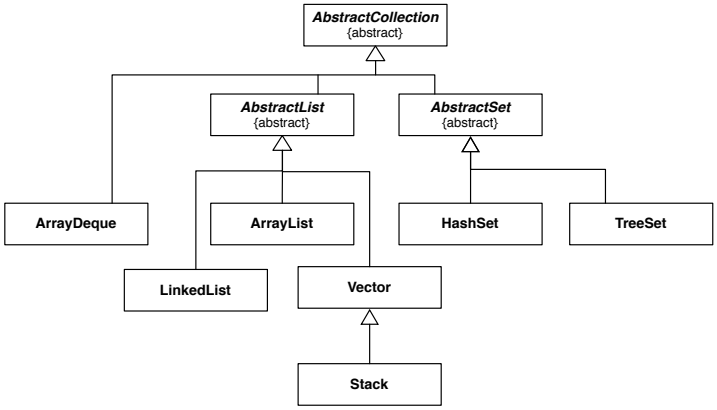
- Das Verhalten von `Map` ist undefiniert, wenn der Schlüssel nach dem Hinzufügen verändert wird
- Schlimmstenfalls werden die Objekte nie mehr wiedergefunden
- Daher sollte man keine veränderbaren Objekte (mutuable objects) als Schlüssel verwenden
- Eine Map darf sich selbst *nicht* als Schlüssel enthalten
- Eine Map sollte sich selbst *nicht* als Wert enthalten

Da das Interface `Map` (analog zu `Set`) in den allermeisten Fällen durch eine Hash-Tabelle realisiert wird, darf man auf keinen Fall den Inhalt des Objektes, das als Schlüssel verwendet wurde, verändern, solange es in der Map gespeichert ist. Die Begründung für diese Regel wurde bereits beim `Set` gegeben und soll hier nicht wiederholt werden. Den Inhalt des Wertes darf man aber beliebig ändern, ohne dass dies Auswirkungen hätte.

Kapitel 6

Collection Klassen

6.1 Klassen [52]



6.2 Klassen und Implementierung [53]

Interface	Hash-Tabelle	Array	B-Baum	Verkettete Listen
Set	HashSet	–	TreeSet	–
List	–	ArrayList	–	LinkedList
Deque	–	ArrayDeque	–	LinkedList
Map	HashMap	–	TreeMap	–

- `ArrayList` implementiert das `List`-Interface basierend auf Arrays `ArrayList`
  - Wahlfreier Zugriff über `get(int index)` ist billig
  - Anfügen neuer Elemente ist bedingt performant
  - Einfügen von Elementen ist teuer
- `LinkedList` verwendet verkettete Listen `LinkedList`
  - Wahlfreier Zugriff über `get(int index)` ist teuer
  - Anfügen neuer Elemente ist billig
  - Einfügen von Elementen ist billig

- `HashSet` implementiert das `Set`-Interface basierend auf den Hash-Werten `HashSet`
  - Grundlegende Operationen (`add`, `remove`, `contains`) sind O(1)
  - Daten sind nicht geordnet
  - Verwendet `equals()`
- `TreeSet` implementiert das `Set`-Interface basierend auf B-Trees (red-black, selbst-balancierender Binärbaum) `TreeSet`
  - Grundlegende Operationen (`add`, `remove`, `contains`) sind O(log(n))
  - Daten sind geordnet und sortiert
  - Verwendet `compareTo()` statt `equals()`

Es gibt eine ganze Reihe unterschiedlicher Implementierungen für die Interfaces des Collection-APIs. Bezüglich der Schnittstelle sind diese Klassen gleich; welche man verwendet hängt vom Anwendungszweck ab und welche Operationen am häufigsten vorkommen. Die folgende Tabelle fasst die Eigenschaften kurz für die Implementierungen von `List`zusammen:

Operation	ArrayList	LinkedList
anfügen	bedingt schnell	schnell
einfügen	langsam	schnell
Zugriff per Index	schnell	langsam
iterieren	schnell	schnell

Bei den Implementierungen von `Set` ist das `HashSet` beim Lesen und Schreiben schneller als das `TreeSet`, muss dafür aber über den Umweg einer Liste sortiert werden. Beim `TreeSet` ergibt sich die Sortierung von alleine.

6.3 Die Sache mit null [56]

Ob man einer Collection `null` hinzufügen darf ist implementierungsabhängig

- `ArrayList` und `LinkedList` erlauben `null` als Wert für beliebig viele Einträge
- `TreeSet` erlaubt nur dann `null` als Wert, wenn ein expliziter Comparator angegeben wird, bei natural ordering ist `null` verboten
- `HashSet` erlaubt `null` grundsätzlich
- `HashMap` erlaubt `null` als Schlüssel und als Wert
- `Hashtable` verbietet `null` als Schlüssel und Wert
- `TreeMap` erlaubt nur dann `null` als Schlüssel, wenn ein expliziter Comparator angegeben wird, bei natural ordering ist `null` verboten

Da die Regeln nur schwer zu merken sind und eigentlich nur das Interface als Maßstab benutzt werden sollte, nicht aber die Implementierung, ist generell davon abzuraten, `null` in einer Collection zu speichern. Braucht man ein Element, das das „Nichts“ repräsentiert, erstellt man

einfach ein spezielles Objekt, das für den fehlenden Wert steht und prüft dann mit == ob dieses oder ein normales Objekt vorliegt.

6.4 Die Klasse Collections [57]

Enthält Hilfsmethoden für den Umgang mit Collections

Auswahl von Methoden

- `emptyList()`, `emptyMap()`, `emptySet()`
- `fill(List<? superT>, T)`
- `max(Collection<? extends T)`, `min(...)`
- `List<T> nCopies(int, T)`
- `reverse(List<T>)`
- `shuffle(List<?>)`
- `synchronizedCollection(Collection<T>)`, `synchronizedList(List<T>)`, ...
- `unmodifiableCollection(Collection<T>)`, `unmodifiableList(List<T>)`, ...

Die Klasse `Collections` enthält eine ganze Reihe von Hilfsmethoden, die man aus Gründen der Übersichtlichkeit und Allgemeingültigkeit nicht in den Interfaces untergebracht hat. Hierzu gehören Methoden, um die Elemente von Liste zu mischen, zu sortieren, umzukehren etc.

`Collections`

6.5 Das alte Collections-API [59]

- Es gibt ein altes und ein neues Collections-API
- Aus Gründen der Kompatibilität sind noch die alten Klassen noch vorhanden
  - `Hashtable` — `Hashtable`
  - `Vector` — `Vector`
  - `Enumeration` — `Enumeration`
- Das alte API sollte *nicht* mehr verwendet werden!

Das neue Collections-API ist die Obermenge des alten und bietet alle Funktionen, die das alte API anbot. Es hat darüber hinaus aber noch mehr Fähigkeiten und ist sauberer entworfen. Daher sollte man das alte Collection-API nicht mehr verwenden.

Kapitel 7

Ordnen und Sortieren

7.1 Geordnet vs. sortiert [61]

Wir müssen unterscheiden

- **geordnet** (ordered) → Die Reihenfolge der Elemente hat eine bestimmte Bedeutung und wird von der Collection erhalten — geordnet
- **sortiert** (sorted) → Die Reihenfolge der Elemente folgt einem Sortierkriterium — sortiert
- Eine sortierte Liste ist immer geordnet
- Eine geordnete Liste kann (muss aber nicht) sortiert sein

Für das weitere Verständnis ist es wichtig, dass man sauber zwischen den Begriffen *geordnet* und *sortiert* unterscheidet. Man kann eine Collection nur dann sortieren, wenn sie geordnet ist, d. h. die Reihenfolge der Elemente über die Zeit erhält.

Für eine *Sortierung* benötigt man zwingend ein **Sortierkriterium**, d. h. es muss eine Funktion geben, die angibt, ob ein Element A vor oder nach einem Element B einsortiert werden soll. Diese Funktion wird in Java über die Interfaces `Comparator` und `Comparable` realisiert (s. u.). — Sortierkriterium

Für eine *Ordnung* ist es nur nötig, dass man den Elementen einen Index zuordnen kann, der das Element stabil bezeichnet – zumindest solange man keine neuen Elemente einfügt.

7.2 Comparable und Comparator [62]

- Klassen können das `Comparable<T>`-Interface implementieren, um ihre **natürliche (Sortier-)Reihenfolge** (natural ordering) festzulegen — natürliche (Sortier-)Reihenfolge
  `int compareTo(T object)`
- Mit `Comparator<T>` kann man explizit eine Klasse schreiben, die Reihenfolge bzw. Sortierung von Objekten einer anderen Klasse bestimmt
  `int compare(T o1, T o2)`
- Rückgabewerte der Methoden: < 0, 0, > 0
- Zusammenhang mit `equals()`
  `(x.compareTo(y)== 0)== (x.equals(y))`

Java realisiert die Sortierung von Elementen über zwei Interfaces:

- `Comparable` kann von einer Klasse implementiert werden, um die *natürliche Sortierreihenfolge* anzugeben. Jede Klasse kann nur *genau eine* solche besitzen. Sie gibt an, wie die Objekte im Normalfall sortiert werden sollen. Für Strings wäre dies die übliche alphanumerische Sortierung „Albert“, Alfons„, “Alfred„, “Beate"… `Comparable`
- `Comparator` erlaubt es, zusätzlich zu der natürlichen Reihenfolge, noch beliebig viele andere *externe Reihenfolgen* anzugeben. Das Interface wird nicht von der zu sortierenden Klasse selbst, sondern von einer anderen Klasse implementiert. Für jede Reihenfolge wird hier eine eigene Klasse geschrieben, die das Interface implementiert. `Comparator`

Beide Interfaces implementieren Methoden, um zwei Objekte miteinander zu vergleichen. Im Falle von `Comparable` über `compareTo` wird `this` mit einem übergebenen Objekt verglichen. Bei `Comparator` bekommt `compare` zwei übergebene Objekte. Der Rückgabewert der Methode gibt an, wie die Reihenfolge der beiden Objekte sein soll:

- `< 0` das erste Objekt (bzw. `this`) ist „kleiner“, also vor dem zweiten einzusortieren. Häufig gibt man an dieser Stelle einfach ein `-1` zurück, es sind aber alle Werte `< 0` zulässig.
- `0` die beiden Objekte sind „gleich“
- `> 0` das erste Objekt ist „größer“, also nach dem zweiten einzusortieren. Hier wird ebenfalls häufig einfach ein `1` zurückgegeben. Es sind aber größere Werte möglich.

Der Grund warum man beliebige positive und negative Werte zurückgeben darf liegt darin, dass sich damit viele Vergleiche einfacher durchführen lassen. So kann man z. B. das absteigende Sortieren von ganzen Zahlen einfach dadurch erreichen, dass man die eine von der anderen abzieht.

7.3 Beispiel: Comparable [63]

```
public class Student implements Comparable<Student> {
    private int matrikelNummer;
    private String name;

    public Student(int matrikelNummer, String name) {
        this.matrikelNummer = matrikelNummer;
        this.name = name;
    }

    public String toString() {
        return matrikelNummer + " - " + name;
    }
}
```

compareTo-Methode für Student

```
    public int compareTo(Student o) {
        if (o.matrikelNummer == matrikelNummer) {
```

```
            return 0;
        }
        else if (o.matrikelNummer > matrikelNummer) {
            return -1;
        }
        else {
            return 1;
        }
    }
}
```

Der Source-Code ist unvollständig, da noch die `hashCode()` und `equals()`-Methode angegeben werden müssen. Aus Platzgründen wurden sie hier weggelassen.

Da es bei der `compareTo`-Methode nur darauf ankommt, ob der Rückgabewert <, > oder = Null ist, kann man die obige Methode noch deutlich kürzer schreiben:

compareTo-Methode für Student

```
    public int compareTo(Student o) {
        return matrikelNummer - o.matrikelNummer;
    }
}
```

7.4 Beispiel: Comparable (mit ArrayList) [65]

```
List<Student> studenten = new ArrayList<>();

studenten.add(new Student(4411, "Franz Müller"));
studenten.add(new Student(5711, "Albert Meier"));
studenten.add(new Student(4711, "Xaver Meier"));

Collections.sort(studenten);

for (Student student : studenten) {
    System.out.println(student);
}
```

```
4411 - Franz Müller
4711 - Xaver Meier
5711 - Albert Meier
```

7.5 Beispiel: Comparable (mit TreeSet) [66]

```
Set<Student> studenten = new TreeSet<>();

studenten.add(new Student(4411, "Franz Müller"));
studenten.add(new Student(5711, "Albert Meier"));
studenten.add(new Student(4711, "Xaver Meier"));

for (Student student : studenten) {
    System.out.println(student);
}
```

```
4411 - Franz Müller
4711 - Xaver Meier
5711 - Albert Meier
```

7.6 Beispiel: Comparator [67]

```
public class Student {

    int matrikelNummer;
    String name;

    public Student(int matrikelNummer, String name) {
        this.matrikelNummer = matrikelNummer;
        this.name = name;
    }

    public String toString() {
        return matrikelNummer + " - " + name;
    }
}
```

```
public class StudentComparator implements Comparator<Student> {

    public int compare(Student o1, Student o2) {
        if (o2.matrikelNummer == o1.matrikelNummer) {
            return 0;
        }
        else if (o2.matrikelNummer > o1.matrikelNummer) {
            return -1;
        }
        else {
            return 1;
```

```
        }
    }
}
```

7.7 Beispiel: Comparator (mit ArrayList) [69]

```
List<Student> studenten = new ArrayList<>();

studenten.add(new Student(4411, "Franz Müller"));
studenten.add(new Student(5711, "Albert Meier"));
studenten.add(new Student(4711, "Xaver Meier"));

Collections.sort(studenten, new StudentComparator());

for (Student student : studenten) {
    System.out.println(student);
}
```

```
4411 - Franz Müller
4711 - Xaver Meier
5711 - Albert Meier
```

7.8 Beispiel: Comparator (mit TreeSet) [70]

```
Set<Student> studenten = new TreeSet<>(new StudentComparator());

studenten.add(new Student(4411, "Franz Müller"));
studenten.add(new Student(5711, "Albert Meier"));
studenten.add(new Student(4711, "Xaver Meier"));

for (Student student : studenten) {
    System.out.println(student);
}
```

```
4411 - Franz Müller
4711 - Xaver Meier
5711 - Albert Meier
```

7.9 Methoden von SortedSet [71]

- `Comparator<? super E> comparator()` – Gibt den Comparator zurück, mit dem die Elemente sortiert werden oder `null` wenn die natürliche Ordnung verwendet wird
- `SortedSet<E> subSet(E fromElement, E toElement)` – Erzeugt eine Teilmenge von `fromElement` (inklusive) bis `toElement` (exklusiv)
- `SortedSet<E> headSet(E toElement)` – Erzeugte eine Teilmenge vom Anfang bis `toElement` (exklusiv)
- `SortedSet<E> tailSet(E fromElement)` – Erzeugt eine Teilmenge von `fromElement` (inklusive) bis zum Ende
- `E first()` – Gibt das erste Element zurück
- `E last()` – Gibt das letzte Element zurück

`SortedSet`

Das Interface `SortedSet` beschreibt ein spezielles `Set`, bei dem die Objekte bereits beim Einfügen sortiert werden.

Existiert eine Sortierung, kann man Operationen der Form „von Element C, bis Element G“ durchführen, da durch die Sortierung festgelegt ist, welche Elemente zwischen diesen beiden Grenzen liegen können. Deswegen sind hier dann die Methoden `subSet`, `headSet` und `tailSet` möglich.

7.10 Beispiel: SortedSet [72]

```
SortedSet<String> set = new TreeSet<>();

set.add("Seeler");
set.add("Mueller");
set.add("Zander");
set.add("Beckenbauer");
set.add("Schumacher");

System.out.printf("Elemente: %s%n", set);
System.out.printf("Erstes Element: %s%n", set.first());
System.out.printf("Letztes Element: %s%n", set.last());
```

```
Elemente: [Beckenbauer, Mueller, Schumacher, Seeler, Zander]
Erstes Element: Beckenbauer
Letztes Element: Zander
```

Da das `TreeSet` die Elemente bereits beim Einfügen sortiert, entfällt in diesem Beispiel eine explizite Sortieroperation: die Daten liegen im `SortedSet` immer sortiert vor.

7.11 Beispiel: Sortieren mit Collections.sort [73]

```
List<String> list = new ArrayList<>();
list.add("Seeler");
list.add("Mueller");
list.add("Zander");
list.add("Beckenbauer");
list.add("Schumacher");

Collections.sort(list);
System.out.println(list);

Collections.sort(list, new ReverseComparator());

System.out.println(list);
```

```
[Beckenbauer, Mueller, Schumacher, Seeler, Zander]
[Zander, Seeler, Schumacher, Mueller, Beckenbauer]
```

```
class ReverseComparator implements Comparator<String> {

    public int compare(String o1, String o2) {
        return o1.compareTo(o2) * -1;
    }
}
```

Der `ReverseComparator` dreht einfach die Sortierreihenfolge der normalen `compaerTo()`-Methode von `String` um, sodass alle Elemente rückwärts sortiert werden.

Kapitel 8

Enumerationen und Collections

8.1 EnumSet und EnumMap [76]

- Es gibt spezielle Collection-Klassen für Enumerationen
  - `EnumSet` EnumSet
  - `EnumMap` EnumMap
- `EnumSet` dient der effizienten Speicherung von Enumerationen (interne Darstellung als Bit-Felder und nicht als Array)
- Bieten zusätzliche Methoden, um Mengen von Konstanten zu verwalten (Differenzmengen etc.)

Da Enumerationen in Java durch vollwertige Objekte realisiert werden, könnte man natürlich zu deren Speicherung normale Collections verwenden. Diese Speicherung ist aber nicht sonderlich effizient, weil man pro Enum-Konstante ein Referenz von 4 oder 8 Byte benutzen muss. Deutlich effizienter ist die Ablage als Bit-Feld. Hier wird einfach für jede mögliche Konstante einer Enumeration ein Bit reserviert. Ist das Bit gesetzt, ist die Konstante im Set vorhanden, ist es nicht gesetzt, fehlt sie. Anstatt 8 Byte benötigt man dann für jede mögliche Konstante nur noch ein Bit. Unsere Enumeration `Planet` mit 8 Planeten-Konstanten kann daher in einem einzigen Byte gespeichert werden, anstatt 64 Byte zu verbrauchen, wie es in einem normalen Set nötig wäre.

Zusätzlich zu der Speichereffizienz bieten die speziellen Klassen für Enumerationen noch Methoden an, um Mengenoperationen durchzuführen. Diese werden später noch vorgestellt.

8.2 Beispiel: Todesstern mit EnumSet [77]

```
import java.util.EnumSet;
import static pr2.Planet.*;

public class Todesstern {

    public void zerstoere(EnumSet<Planet> planeten) {
        for (Planet planet : planeten) {
```

```
            richteTodesstrahlAus(planet);
        }
    }

    public static void main(String[] args) {
        Todesstern ts = new Todesstern();
        ts.zerstoere(EnumSet.of(MARS, VENUS, JUPITER));
    }

    ...
}
```

Das Beispiel zeigt einen Todesstern, der in einem Durchlauf mehrere Planeten vernichten kann. Zu diesem Zweck wird der Methode `zerstoere()` ein `EnumSet` mit allen Planeten übergeben, die er vernichten soll.

Praktisch ist die Möglichkeit über die statische Methode `of()` der Klasse `EnumSet` die Konstanten direkt anzugeben, die eingefügt werden sollen. Ohne diese Methode müsste man deutlich umständlicher schreiben:

```
EnumSet<Planet> planeten = new EnumSet<>();
planeten.add(Planet.MARS);
planeten.add(Planet.VENUS);
planeten.add(Planet.JUPITER);
```

8.3 Beispiel: Hilfsmethoden auf EnumSet [78]

```
EnumSet<Planet> planeten = EnumSet.allOf(Planet.class);
System.out.println("allOf: " + planeten);

EnumSet<Planet> leer = EnumSet.noneOf(Planet.class);
System.out.println("noneOf: " + leer);

EnumSet<Planet> dieAnderen = EnumSet.complementOf(EnumSet.of(Planet.ERDE));
System.out.println("complementOf: " + dieAnderen);

EnumSet<Planet> aeussere = EnumSet.range(Planet.MARS, Planet.NEPTUN);
System.out.println("range: " + aeussere);
```

```
allOf: [MERKUR, VENUS, ERDE, MARS, JUPITER, SATURN, URANUS, NEPTUN]
noneOf: []
complementOf: [MERKUR, VENUS, MARS, JUPITER, SATURN, URANUS, NEPTUN]
range: [MARS, JUPITER, SATURN, URANUS, NEPTUN]
```

Das Beispiel zeigt weitere nützliche Methoden des `EnumSet`:

- `allOf` liefert ein Set, in dem alle Konstanten der Enumeration enthalten sind
- `noneOf` liefert ein leeres EnumSet
- `complementOf` liefert ein Set mit allen Konstanten der Enumeration bis auf die als Parameter übergebenen
- `range` liefert ein Set mit allen Konstanten „zwischen“ den beiden Parametern

8.4 Beispiel: Ohne EnumSet (old school) [79]

```
public class Auszeichnungen {
    public static final int BOLD = 0x01;
    public static final int ITALIC = 0x02;
    public static final int UNDERLINE = 0x04;
    public static final int STRIKEOUT = 0x08;
    public static final int OUTLINED = 0x10;
}
```

Bevor es Enumerationen und die dazugehörigen Collection-Klassen gab, musste man sich eine effiziente Speicherung selbst bauen. Dazu verwendete man `int`-Werte, die Zweierpotenzen waren.

```
Textprogramm t = new Textprogramm();
t.setAuszeichnung(BOLD | UNDERLINE | STRIKEOUT);
```

Der Verwender konnte dann die gewünschten Werte durch ein bitweises Oder (`|`) verknüpfen. Hierdurch wurden die Bits gesetzt, die den Konstanten entsprachen.

```
    public void setAuszeichnung(int auszeichnung) {
        if ((auszeichnung & BOLD) > 0) {
            System.out.println("Fett");
        }
        if ((auszeichnung & ITALIC) > 0) {
            System.out.println("Kursiv");
        }
        if ((auszeichnung & UNDERLINE) > 0) {
            System.out.println("Unterstrichen");
        }
        if ((auszeichnung & STRIKEOUT) > 0) {
            System.out.println("Durchgestrichen");
        }
        if ((auszeichnung & OUTLINED) > 0) {
            System.out.println("Umriss");
        }
```

```
    }
```

Durch Anwenden einer Bitmaske konnte dann die Methode teste, welche Bits in dem übergebenen `int`-Wert gesetzt waren.

8.5 Beispiel: Mit EnumSet [82]

```
public enum Auszeichnungen {
    BOLD, ITALIC, UNDERLINE, STRIKEOUT, OUTLINED;
}
```



```
Textprogramm t = new Textprogramm();
t.setAuszeichnung(EnumSet.of(BOLD, UNDERLINE, STRIKEOUT));
```



```
    public void setAuszeichnung(EnumSet<Auszeichnungen> auszeichnung) {
        if (auszeichnung.contains(BOLD)) {
            System.out.println("Fett");
        }
        if (auszeichnung.contains(ITALIC)) {
            System.out.println("Kursiv");
        }
        if (auszeichnung.contains(UNDERLINE)) {
            System.out.println("Unterstrichen");
        }
        if (auszeichnung.contains(STRIKEOUT)) {
            System.out.println("Durchgestrichen");
        }
        if (auszeichnung.contains(OUTLINED)) {
            System.out.println("Umriss");
        }
    }
```



Mit Enumerationen und `EnumSet` fällt das Beispiel deutlich übersichtlicher und einfacher aus. Vor allem gibt es deutlich weniger Fehlerquellen.

Kapitel 9

Collections und Nebenläufigkeit

9.1 ConcurrentModificationException [86]

fail-fast

ConcurrentModification Exception

- Collections und Maps sind **fail-fast**: Sie entdecken konkurrierende Zugriffe und werfen eine Ausnahme (**ConcurrentModification Exception**) wenn eine solche vorkommt
- Daher darf man während einer Iteration nur über den Iterator die Collection verändern, nicht aber am Iterator vorbei
- Direkte Änderungen während des Iterierens sind verboten
- Man sollte aber die `ConcurrentModificationException` nicht fangen und sich nicht darauf verlassen, dass sie geworfen wird

Ein Iterator, der über eine Collection läuft, stellt eine Momentaufnahme der Daten in dem Collection-Objekt dar. Wenn die Collection während der Iteration verändert würde, könnte es zu ungültigen Zuständen kommen. Man stelle sich z. B. vor, der Iterator wollte ein Objekt zurückgeben, dass genau in diesem Augenblick aus der Collection entfernt wird. Oder es wird hinter der aktuellen Iterator-Position ein Objekt in die Collection eingefügt.

Aus diesem Grund ist das *Verändern der Collection während der Iteration grundsätzlich verboten.* Die Collection-Klassen werfen daher eine Ausnahme, wenn dies trotzdem versucht wird.

Für die nebenläufige Programmierung existieren spezielle Collection-Klassen, die parallele Veränderungen und Lesevorgänge zulassen. Diese werden später noch vorgestellt.

9.2 Beispiel: ConcurrentModificationException [87]

```
List<String> list = new ArrayList<>();
Collections.addAll(list, "A", "B", "C", "D", "E", "X", "F");

for (String element : list) {
    if ("X".equals(element)) {
        list.remove(element); // FEHLER!!
    }
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.AbstractList$Itr.checkForComodification(
          AbstractList.java:372)
    at java.util.AbstractList$Itr.next(AbstractList.java:343)
    at pr2.CMEDemo.main(ConcurrentModificationException.java:14)
```

Das Entfernen des Elements aus der Collection während der Iteration durch `list.remove(element)` führt zu einer Ausnahme.

Der im Beispiel gewählte Ansatz ist sowieso nicht sinnvoll, da alle Collections eine Methode `remove` haben, mit der man ohne Verwendung eines Iterators Objekte aus der Collection entfernen kann.

9.3 Thread-Safety [88]

- Die modernen Collection Implementierungen (`ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`) sind nicht thread-safe
- Die alten Implementierungen (`Vector`, `Hashtable`) sind thread-safe, da alle Methoden synchronisiert sind
- Man kann jedes Collection-Objekt mit `Collections.synchronized...` in ein threadsicheres umwandeln

threadsicher

Wenn eine Collection von mehreren Threads gleichzeitig benutzt werden soll, muss sie **threadsicher** (thread-safe) sein. Wie im Kapitel zu Threads erläutert wird, müssen die Methoden mit `synchronized` als kritische Abschnitte definiert werden. Im alten Collection-API sind alle Methoden entsprechend markiert, im neuen nicht. Der Grund liegt darin, dass der Aufruf einer synchronisierten Methode aufwändiger ist als der einer nicht-synchronisierten. Da die meisten Collections von nur einem Thread benutzt werden, wäre es unnötig die Methoden grundsätzlich zu synchronisieren.

Benötigt man eine threadsichere Collection, kann man diese einfach über Klasse `Collections` anfordern. Dort gibt es für alle Interfaces eine Methode `sychronized...` (z. B. `synchronizedSet`), die eine threadsichere Variante der übergebenen Collection zurückliefert.

9.4 ConcurrentHashMap<K, V> [89]

`ConcurrentHashMap`

ConcurrentHashMap

- Verwendet deutlich bessere Locking-Strategie als `Collections.synchronizedMap(...)`
- Iteratoren werfen keine `ConcurrentModificationException`
- Besitzt zusätzliche Methoden für das check-then-act-Pattern, um externe Synchronisation zu vermeiden

9.5 ConcurrentHashMap<K, V> [90]

Methoden

- `V putIfAbsent(K key, V value)` – Fügt `value` nur hinzu, wenn für den Schlüssel `k` noch kein Eintrag vorhanden ist
- `boolean remove(Object key, Object value)` – Entfernt den Schlüssel `key` nur dann, wenn er mit dem gegebenen Wert `value` verbunden ist
- `boolean replace(K key, V oldValue, V newValue)` – Ersetzt den Wert für den Schlüssel `key` nur wenn `key` vorhanden ist und auf `oldValue` zeigt
- `V replace(K key, V value)` – Ersetzt den Wert `value` für den Schlüssel `key` nur dann, wenn `key` bereits vorhanden ist

9.6 Beispiel: ConcurrentHashMap [91]

```
if (!map.containsKey(key)) {
    map.put(key, value);
}
else {
    value = map.get(key);
}


value = map.putIfAbsent(key, value);
```

9.7 CopyOnWriteArrayList<E> [92]

`CopyOnWriteArrayList` `CopyOnWriteArrayList`

- Optimierte Liste für Szenarien, in denen viele Threads von einer Liste lesen, sie aber nur selten geschrieben wird
- Normale Synchronisation würde hier erhebliche Geschwindigkeitseinbussen bedeuten
- Bei jeder Schreiboperation wird eine Kopie der Liste angelegt
- Iteratoren werfen keine `ConcurrentModificationException`

Index

assoziatives Array, 19

Collection-API, 3
Collections, 2
ConcurrentModification, 37

Duplikate, 11

Exception, 37

fail-fast, 37
foreach, 6

geordnet, 26

Hash, 19

Indices, 11
iterieren, 4

konstanter Zeit, 19

List, 15

natürliche (Sortier-)Reihenfolge, 26

Ordnung, 11

Schlüssel, 19
Schlüssel-Wert-Paaren, 19
Sortierkriterium, 26
sortiert, 26

threadsicher, 38

Wert, 19
Wörterbüchern, 19

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Streams



Wintersemester 2025/2026

Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**

1.1 Fragen [5] . . . 1
1.2 Begriffe [6] . . . 1
1.3 Links zur Java-Insel [7] . . . 1

2 Motivation **2**

2.1 Das grundlegende Problem [9] . . . 2
2.2 Streams [10] . . . 3
2.3 Beispiel für Iteration mit Stream [11] . . . 3

3 Grundlegende Operationen **5**

3.1 Arten von Operationen auf Streams [13] . . . 5
3.2 Konzepte des Stream-APIs [14] . . . 6
3.3 Collect [15] . . . 7
3.4 Filter [17] . . . 8
3.5 Map [19] . . . 9
3.6 Reduce [23] . . . 11
3.7 flatMap [26] . . . 12
3.8 Optional [28] . . . 13
3.9 Maximum und Minimum [29] . . . 14
3.10 Beispiel: Kombinierte Operationen [30] . . . 14
3.11 Spezialisierte Streams [35] . . . 17
3.12 Beispiel: IntStream [36] . . . 18
3.13 Beispiel: Umwandlung zu einem IntStream [37] . . . 18

4 Fortgeschrittene Techniken **19**

4.1 Reihenfolge der Elemente im Stream [39] . . . 19
4.2 Collector [40] . . . 19
4.3 Daten aufteilen [43] . . . 20
4.4 Gruppieren von Daten [44] . . . 21
4.5 Erzeugung von Strings [45] . . . 21
4.6 Parallelisierung [46] . . . 22

Index **i**

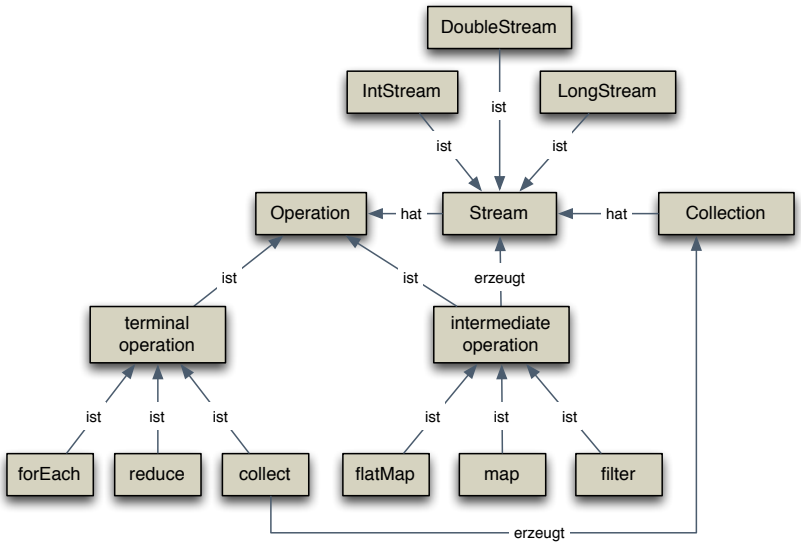
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie kann ich Daten funktional verarbeiten?
- Wie kann ich schnell auf großen Datenmengen arbeiten?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 18.4 Java Stream-API
- 18.5 Einen Stream erzeugen
- 18.6 Terminale Operationen
- 18.7 Intermediäre Operationen

Kapitel 2

Motivation

2.1 Das grundlegende Problem [9]

- Collections erlauben Objektmengen zu verwalten
- Bearbeiten von Collections erfolgt über **externe Iteration** (external iteration) externe Iteration
  - fehleranfällig
  - unübersichtlich
  - nicht parallelisierbar

```
List<String> namen = Arrays.asList("Franz", "Peter", "Jakob", "Hans");
Iterator<String> it = namen.iterator();

while (it.hasNext()) {
    String name = it.next();
    System.out.println(name);
}
```

Externe Iteration ist ein typisches Vorgehen bei imperativen Programmiersprachen. Das Problem liegt darin, dass der Quelltext relativ weit von der Absicht (Intention) entfernt ist. Im vorliegenden Beispiel ist die Intention: „Gib den Wert aller Elemente der Collection aus“. Diese Absicht geht aber im Quelltext weitgehend unter und wird durch eine Kontrollstruktur und zwei temporäre Variablen verschleiert.

Man kann das Beispiel mithilfe der foreach-Loop noch etwas kompakter schreiben, das grundlegende Problem bleibt aber.

```
List<String> namen = Arrays.asList("Franz", "Peter", "Jakob", "Hans");

for (String name : namen) {
    System.out.println(name);
}
```

2.2 Streams [10]

Streams

- Java 8 führt eine neue Art ein, mit Collections zu arbeiten: **Streams**

stream()

- Über die Methode `stream()` kann man zu jeder Collection einen Stream erhalten
- Streams folgenden den Prinzipien der *Funktionalen Programmierung*, d. h. man teilt dem Stream über eine Funktion mit, was geschehen soll

interne Iteration

- Streams erlauben eine **interne Iteration** (internal iteration)
  - Filtern von Einträgen
  - Zusammenführen von Collections
  - Aufspalten von Collections
  - Umwandeln zwischen verschiedenen Collection-Arten
  - …

Man darf die Streams für Collections nicht mit den gleichnamigen Klassen für die Eingabe- und Ausgabe verwechseln. Es gibt zwar einige Gemeinsamkeiten aber trotzdem handelt es sich um grundverschiedene Konzepte, die weitgehend nur den Namen teilen.

Streams sind eng mit Lambdas verknüpft, da diese erst die Möglichkeit bieten, einfach eine Funktion an eine Methode des Streams zu übergeben.

2.3 Beispiel für Iteration mit Stream [11]

Mit Lambda

```
List<String> namen = Arrays.asList("Franz", "Peter", "Jakob", "Hans");

namen.stream().forEach(name -> System.out.println(name));
```

Mit Methodenreferenz

```
List<String> namen = Arrays.asList("Franz", "Peter", "Jakob", "Hans");

namen.stream().forEach(System.out::println);
```

Das obige Beispiel zeigt die einfachste Form einer Stream-Operation, eine interne Iteration, bei der auf jedem Element die übergebene Funktion angewandt wird.

Theoretisch hätte man Streams in Java bereits früher, ohne Lambdas einführen und stattdessen innere Klassen verwenden können. Da hierdurch die Syntax aber extrem unübersichtlich wird, sind Streams nur zusammen mit Lambdas wirklich nutzbar. Ohne Lambdas müsste man das Beispiel wie folgt schreiben:

```
List<String> namen = Arrays.asList("Franz", "Peter", "Jakob", "Hans");
```

```
namen.stream().forEach(new Consumer<String>() {
    public void accept(String name) {
        System.out.println(name);
    }
});
```

Es ist offensichtlich, dass hier der Gewinn einer internen Iteration durch die verworrene Syntax mehr als wettgemacht wird. Obwohl das Beispiel also semantisch vollkommen identisch mit dem vorhergehenden ist, ist es zu unübersichtlich, um nützlich zu sein.

Kapitel 3

Grundlegende Operationen

3.1 Arten von Operationen auf Streams [13]

- Streams sind optimiert und versuchen so wenig wie möglich Iterationen durchzurühren
- Man unterscheidet bei den Stream-Operationen
  - **intermediate operations**: beschreiben, was mit den Daten zu tun ist, führen aber noch nicht zu einer Iteration oder Aktion (z. B. `filter()`)
  - **terminal operations**: erzeugen ein Ergebnis und benötigen eine Iteration (z. B. `forEach()` oder `collect()`)
- Nach einer *terminal operation* wird der Stream geschlossen und kann nicht mehr verwendet werden

intermediate operations

terminal operations

Die *intermediate operations* bezeichnet man als **lazy** (faul), die *terminal operations* als **eager** (ehrgeizig). Die intermediate operations geben einen Stream zurück, sodass man über Method-Chaining weitere Operationen aufrufen kann.

lazy
eager

Dass die Auswertung wirklich erst bei einer terminal operation durchgeführt wird, sieht man an folgendem Beispiel, das keine Ausgabe erzeugt, weil keine terminal operation aufgerufen wird:

Nur intermediate operations



```
List<String> namen = Arrays.asList(
        "Anton", "Egon", "Alfred", "Barbara", "Klaus", "Herbert");

namen.stream()
        .filter(name -> {
            System.out.println("Filter 1: " + name);
            return name.startsWith("A");
        })
        .filter(name -> {
            System.out.println("Filter 2: " + name);
            return name.charAt(1) == 'n';
        });
```

Erst das Hinzufügen einer entsprechenden Operation (z. B. `forEach()`) führt hier zu einer Ausgabe.

Abschluss mit einer terminal operation



```
 namen.stream()
       .filter(name -> {
           System.out.println("Filter 1: " + name);
           return name.startsWith("A");
       })
       .filter(name -> {
           System.out.println("Filter 2: " + name);
           return name.charAt(1) == 'n';
       })
       .forEach(System.out::println);
}
```

An der Ausgabe sieht man sehr gut, wie die Operationen optimiert werden:



```
Filter 1: Anton
Filter 2: Anton
Anton
Filter 1: Egon
Filter 1: Alfred
Filter 2: Alfred
Filter 1: Barbara
Filter 1: Klaus
Filter 1: Herbert
```

3.2 Konzepte des Stream-APIs [14]

- Stream-Methoden liefern wieder einen Stream zurück (Ausnahme: *terminal operations*)
- Auf dem Ergebnis können weitere Methoden aufgerufen werden (**method chaining**)
- Collection → Stream über `stream()`-Methode (s. o.)
- Stream → Collection über `collect()`-Methode (s. u.)
- Stream direkt erzeugen über `Stream.of(...)`

method chaining

`stream()`

`collect()`

`Stream.of(...)`



```
Stream<String> stream = Stream.of(
       "Anton", "Egon", "Alfred", "Barbara", "Klaus", "Herbert");
stream.forEach(System.out::println);
```

Das *method chaining* ist in Java durchaus schon vor Streams zu Einsatz gekommen, z. B. bei der Klasse `StringBuilder`.



```
StringBuilder sb = new StringBuilder();
sb.append("Hallo")
  .append(' ')
  .append("Welt");
```

Im Gegensatz zu diesen älteren Klassen ist es aber bei Streams wichtig, dass man es benutzt, da hierdurch die Performance gesteigert wird. Der Grund liegt darin, dass die Klasse bis zum Erreichen einer **terminal operation** keine Werte generieren muss. Dies kann aber nur funktionieren, wenn alle Methoden auf derselben Instanz aufgerufen werden. Aus diesem Grund gilt es als guter Stil, im Zusammenhang mit Streams grundsätzlich die Methoden zu verketten.

terminal operation

Man wird bei Streams immer wie im folgenden Beispiel vorgehen:

Gut: Mit method chaining

```
List<String> namen = Arrays.asList(
              "Anton", "Egon", "Alfred", "Barbara", "Klaus", "Herbert");

namen.stream()
       .filter(name -> name.startsWith("A"))
       .forEach(System.out::println);
```

Das folgende Beispiel gilt als schlechter Stil:

Schlecht: Ohne method chaining / schlechter Stil

```
List<String> namen = Arrays.asList(
       "Anton", "Egon", "Alfred", "Barbara", "Klaus", "Herbert");

Stream<String> aNamen = namen.stream().filter(name -> name.startsWith("A"));
aNamen.forEach(System.out::println);
```

3.3 Collect [15]

- Methode `collect()` sammelt Elemente eines Streams ein (z. B. in `List`)
- Ist eine *terminal operation*
- Parameter gibt an, in welche Datenstruktur die Elemente gepackt werden sollen (Funktion)

collect()

```
Stream<String> stream = Stream.of(
       "Anton", "Egon", "Alfred", "Barbara", "Klaus", "Herbert");

List<String> result = stream
    .filter(name -> name.startsWith("A"))
    .collect(toList());
```

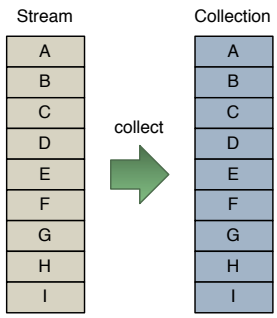
```
System.out.println(result); // -> [Anton, Alfred]
```

Da die `collect()`-Methode sehr umfangreich ist, wird sie später noch einmal behandelt werden. Die Methode `toList()` im Beispiel stammt aus der Klasse `java.util.stream.Collectors` und muss statisch importiert werden, damit das Beispiel wie oben gezeigt funktioniert: `import static java.util.stream.Collectors.toList;`

Ohne diesen Import muss man schreiben:

```
...
List<String> result = stream
    .filter(name -> name.startsWith("A"))
    .collect(Collectors.toList());
```

Die `toList()`-Methode gibt eine Funktion zurück, die der `collect()`-Methode mitteilt, wie sie die Objekte aus dem Stream einsammeln soll. Diese Funktion ist vom Typ `Collector<T, ?, List<T>>`.



Collect-Operation auf einem Stream

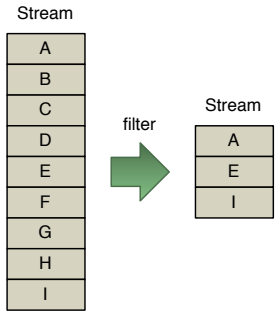
3.4 Filter [17]

filter()

- Methode `filter()` wählt bestimmte Werte aus dem Stream aus
- Ist eine *intermediate operation*
- Parameter ist ein `Predicate<? super T>`
  - `true`: Element soll in den neuen Stream übernommen werden
  - `false`: Element soll nicht übernommen werden

```
List<String> namen = Arrays.asList(
    "Anton", "Egon", "Alfred", "Barbara", "Klaus", "Herbert");
namen.stream()
       .filter(name -> name.startsWith("A"))
       .forEach(System.out::println);
```

```
Anton
Alfred
```



Filter-Operation auf einem Stream

3.5 Map [19]

map()

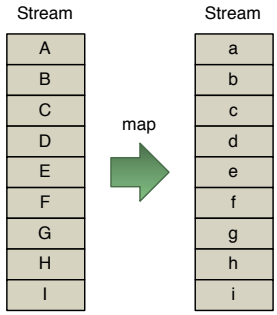
- Methode `map()` konvertiert die Werte aus dem einen Stream in einen neuen Stream
- Ist eine *intermediate operation*
- Parameter legt die Konvertierung fest, ist vom Typ `Function<? super T, ? extends R>`

```
Stream<String> stream = Stream.of(
       "Anton", "Egon", "Alfred", "Barbara", "Klaus", "Herbert");
stream
    .map(name -> name.toLowerCase())
    .forEach(System.out::println); // anton egon ...
```

Man kann das Beispiel noch kürzer schreiben, wenn man für die Map-Operation eine Methodenreferenz verwendet:

```
stream
    .map(String::toLowerCase)
    .forEach(System.out::println); // anton egon ...
```

Bei der Map-Operation handelt es sich um die am meisten genutzt Methode bei den Streams.



Map-Operation auf einem Stream

```
public class Schauspieler {
    private String name;
    private String film;

    public Schauspieler(String name, String film) {
        this.name = name;
        this.film = film;
    }

    public String getName() {
        return name;
    }

    public String getFilm() {
        return film;
    }
}
```

```
Stream<Schauspieler> stream = Stream.of(
        new Schauspieler("John Travolta", "Pulp Fiction"),
        new Schauspieler("Brad Pitt", "Inglourious Basterds"));

stream
    .map(Schauspieler::getName)
    .forEach(System.out::println);
```

Ausgabe

```
John Travolta
Brad Pitt
```

In diesem Beispiel wurde eine Methodenreferenz für die Map-Operation verwendet, um sie kurz zu halten. Statt `Schauspieler::getName` hätte man schreiben können `s -> s.getName()`.

3.6 Reduce [23]

reduce()

- Methode `reduce()` fasst den gesamten Stream zu einem Wert zusammen
- Ist eine *terminal operation*
- Braucht zwei Parameter
  - `T identity`: Startwert der Akkumulation
  - `BinaryOperator<T> accumulator`: Funktion, die beschreibt, wie die Reduktion erfolgen soll

```java
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int summe = stream.reduce(0, (sum, value) -> sum + value);
System.out.println(summe); // -> 55
```

Der Ablauf der `reduce()`-Funktion lässt sich durch folgenden Java-Code beschreiben:

```java
T result = identity;
for (T element : this stream) {
    result = accumulator.apply(result, element)
}
return result;
```

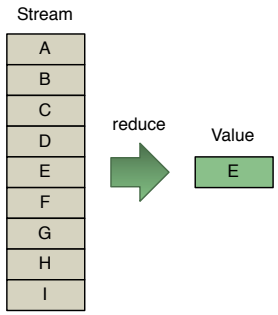
Die übergebene Funktion (hier `(sum, value)-> sum + value`) wird also

- beim ersten Durchlauf mit dem als `identity` übergebenen Wert und dem ersten Element aus dem Stream aufgerufen,
- danach mit dem Rückgabewert des vorhergehenden Aufrufs und dem aktuellen Element aufgerufen.

Die folgende Tabelle zeigt die Werte für die Aufrufe des Lambdas im Beispiel:

Durchlauf	sum	value	Ergebnis
1.	0	1	1
2.	1	2	3
3.	3	3	6
4.	6	4	10
5.	10	5	15
6.	15	6	21
7.	21	7	28
8.	28	8	36
9.	36	9	45
10.	45	10	55

Die hier vorgestellte `reduce()`-Operation geht davon aus, dass die Elemente des Streams und das Ergebnis der Reduktion denselben Typ haben. Wir werden später noch sehen, wie man bei der Reduktion andere Typen als Ergebnis erhalten kann.



Reduce-Operation auf einem Stream

```
Stream<String> stream = Stream.of("A", "B", "C", "D");
String ergebnis = stream.reduce("", (acc, value) -> acc + value);

System.out.println(ergebnis); // -> ABCD
```

Dieses Beispiel verkettet die Elemente des Streams zu einem einzigen String. Später wird noch eine elegantere Methode hierfür vorgestellt.

3.7 flatMap [26]

flatMap()

- Methode `flatMap()` ersetzt einen Wert im Stream mit einem Stream und fügt am Ende alle Streams zu einem einzigen Stream zusammen
- Ist eine *intermediate operation*
- Parameter legt die Konvertierung fest, ist vom Typ `Function<? super T,? extends Stream<? extends R>>`

```
List<String> l1 = Arrays.asList("A", "B", "C");
List<String> l2 = Arrays.asList("D", "E", "F");
List<List<String>> gesamt = Arrays.asList(l1, l2);

List<String> result = gesamt.stream()
       .flatMap(e -> e.stream())
       .collect(toList());
System.out.println(result); // -> [A, B, C, D, E, F]
```

Im Beispiel werden die Daten aus einer Liste, die selbst wieder Liste enthält, in eine einzige Liste zusammengefasst.

Ohne Streams hätte man die im Beispiel gezeigte Funktionalität wie folgt implementieren müssen:

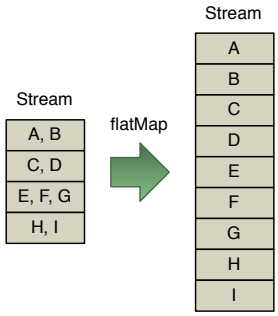
```
List<String> l1 = Arrays.asList("A", "B", "C");
List<String> l2 = Arrays.asList("D", "E", "F");
```

```java
List<List<String>> gesamt = Arrays.asList(l1, l2);
List<String> result = new ArrayList<>();

for (List<String> l : gesamt) {
    for (String s : l) {
        result.add(s);
    }
}

System.out.println(result); // -> [A, B, C, D, E, F]
```

Man sieht hier deutlich die Vorteile, die sich durch die Streams ergeben.



FlatMap-Operation auf einem Stream

3.8 Optional [28]

- Lästiges Problem in Java: Null-Werte
- Stream API gibt nie `null` zurück, sondern falls `null` möglich, ein `Optional` (Typ: `Optional<T>`) `Optional`
- Wichtige Methoden
  - `get()` – gibt den Wert zurück oder wirft Exception, wenn kein Wert vorhanden
  - `isPresent()` – zeigt an, ob Wert vorhanden (`true`)
  - `orElse(T other)` – wenn leer, dann wird `other` zurückgegeben andernfalls der Wert im Optional
  - `ofNullable(T value)` – erzeugt ein Optional aus einem Wert, der `null` sein kann

Die Klasse `Optional` dient dazu, die Fallunterscheidungen, die durch mögliche Null-Werte erzeugt werden, zu vereinfachen. Anstatt jeden Rückgabewert mit `!= null` prüfen zu müssen, bekommt man auf jeden Fall ein gültiges Objekt zurück. Dieses kann man über `isPresent()` fragen, ob ein Wert vorhanden ist oder nicht. Obwohl dies noch keinen Vorteil gegenüber einem Vergleich mit `!= null` bringt, hat man über die Methode `orElse` die Möglichkeit, den Test ganz zu vermeiden. Andere Programmiersprachen kennen dieses Konzept schon lange, so ist zum Beispiel in Ruby `Null` ein vollwertiges Objekt, auf dem man Methoden aufrufen kann.

Da das leere Optional (zu bekommen über `Optional.empty()`) als Singleton realisiert ist, kommt es durch die Verwendung von `Optional` nicht zu einem zusätzlichen Speicherverbrauch.

3.9 Maximum und Minimum [29]

- `max()` und `min()` finden das Maximum/Minimum in einem Stream
- Sind *terminal operations*
- Benötigen einen Comparator, um den Wert zu finden, Typ `Comparator<? super T>`
- Geben ein `Optional` zurück

`max()`
`min()`

```
Stream<String> stream = Stream.of("K", "O", "B", "T", "I");

String max = stream
       .max((a, b) -> a.compareTo(b))
       .orElse("<leer>");

System.out.println(max); // -> T
```

Das Beispiel sucht in dem Stream nach dem String, der gemäß dem angegebenen `Comparator` das Maximum darstellt. Anstatt des Lambdas `(a, b) -> a.compareTo(b)` im Beispiel hätte man die entsprechende Methodenreferenz `String::compareTo` verwenden können.

Das Schreiben eigener Comparatoren kann man bei `max()` und `min()` vermeiden, wenn man auf die Hilfsmethode `Comparator.comparing()` zurückgreift, die es erlaubt durch Angabe einer Funktion, die den Wert eines Objektes extrahiert einen Comparator zu erzeugen. Damit sähe das Beispiel dann wie folgt aus:

```
Stream<String> stream = Stream.of("K", "O", "B", "T", "I");

String max = stream
       .max(Comparator.comparing(a -> a))
       .orElse("<leer>");

System.out.println(max); // -> T
```

3.10 Beispiel: Kombinierte Operationen [30]

```
public class Item {

    private String name;
    private int price;

    public Item(String name, int price) {
        this.name = name;
        this.price = price;
    }
```

```
    public int getPrice() {
        return price;
    }

    public String getName() {
        return name;
    }
}
```

Das Beispiel beginnt mit einer Klasse `Item`, die eine Bestellposition in einer Bestellung repräsentiert (einen Artikel). Jeder Artikel hat nur zwei Informationen: einen Namen und einen Preis (angegeben in Cent).

```
public class Order {

    private List<Item> items = new ArrayList<>();

    public Order(Item ... items) {
        for (Item item : items) {
            this.items.add(item);
        }
    }

    public List<Item> getItems() {
        return this.items;
    }
}
```

Eine Bestellung besteht aus einer Liste von Artikeln. Diese können über den Konstruktor gesetzt werden und über die Methode `getItems()` wieder ausgelesen.

```
List<Order> orders =
    Arrays.asList(
        new Order(
            new Item("Java Insel", 3980),
            new Item("GoTo Java", 2240),
            new Item("Planet der Affen", 1690)
        ),
        new Order(
            new Item("Breaking Dawn", 2130),
            new Item("Lord of the Rings", 1870)
        )
    );
```

Als Datensatz dient eine Liste von Bestellungen (`orders`), die zwei Bestellungen mit drei bzw. zwei Artikeln enthält.

Finde die Namen aller Artikel mit einem Preis > 20 EUR

```
List<String> namen =
    orders.stream()
        .flatMap(e -> e.getItems().stream())
        .filter(e -> e.getPrice() > 2000)
        .map(Item::getName)
        .collect(toList());

System.out.println(namen);
```

```
[Java Insel, GoTo Java, Breaking Dawn]
```

Um alle Artikel in allen Bestellungen zu finden, wird die verschachtelte Struktur der Bestellungen zuerst mit `flatMap` in eine flache Datenstruktur verwandelt. Durch eine Filter-Operation werden mit `filter` diejenigen Positionen extrahiert, deren Preis größer als 2000 Cent ist. Im nächsten Schritt werden durch `map` die Namen der Bestellpositionen extrahiert und abschließend das Ergebnis über `collect` in einer Liste gesammelt.

Würde man versuchen, dasselbe Ergebnis ohne Streams zu erreichen, würde man folgende, stark verschachtelte Schleife benötigen:

Lösung ohne Streams

```
List<String> namen = new ArrayList<>();

for (Order order : orders) {
    for (Item item : order.getItems()) {
        if (item.getPrice() > 2000) {
            namen.add(item.getName());
        }
    }
}

System.out.println(namen);
```

Berechne die Summe der Preise aller Artikel in allen Bestellungen

```
int summe =
    orders.stream()
```

```
        .flatMap(e -> e.getItems().stream())
        .map(Item::getPrice)
        .reduce((a, e) -> a + e)
        .orElse(0);

System.out.println(summe);
```

```
11910
```



Hier wird wieder die Datenstruktur mit `flatMap` in einen flachen Stream von Elementen verwandelt. Dann wird über `map` der Stream der Bestellpositionen durch einen ersetzt, der nur noch die Preise enthält, um diese dann über eine `reduce`-Operation zusammenzuzählen.

Da `reduce` ein Optional und nicht direkt den Wert liefert, wird dieser mit `orElse` ausgelesen. Sollte sich keine Summe bilden lassen, wird als Wert einfach `0` zurückgegeben.

3.11 Spezialisierte Streams [35]

- Speicherung von primitiven Datentypen (`int`, `long`, …) in normalen Collections ist ineffizient
  - ständiges **boxing** und **unboxing**
  - höherer Speicherverbrauch

boxing
unboxing

- Lösung
  - Spezialisierte Streams für primitive Daten: `IntStream`, `LongStream`, `DoubleStream`
  - Spezialisierte Methoden zur Umwandlung zwischen den Welten: `Stream.mapToInt`, `IntStream.mapToObj`, …
  - Spezialisierte funktionale Interfaces für die Lambdas: `IntPredicate`, `IntUnaryOperator`, …

Wie bereits im Kapitel zu den Wrapper-Typen erläutert wurde, wird bei der Zuweisung eines primitiven Datentyps an eine Objektreferenz automatisch eine Konvertierung des primitiven Wertes in ein passendes **Wrapper-Objekt** durchgeführt (**autoboxing**). In die andere Richtung (Zuweisung eines Wrapper-Objektes an einen primitiven Datentyp) findet eine Umwandlung in die andere Richtung statt (**autounboxing**).

Wrapper-Objekt
autoboxing
autounboxing

```
Integer i = 7; // boxing
int k = i; // unboxing
```



Diese Operationen sind rechenintensiv und daher zu vermeiden, wenn die Eingaben und Ausgaben einer Berechnung ohnehin primitiv sind.

Ein weiteres Problem ist der Speicherverbrauch von Wrapper-Objekten. Während ein `int` nur 32 Bit belegt, benötigt ein `Integer`-Objekt ein vielfaches (gespeicherter Wert, Referenz auf das Class-Objekt, Objekt-Header etc.).

3.12 Beispiel: IntStream [36]

```
IntStream stream = IntStream.of(8, 9, 12, 22, 2, 3, 44, 11, 7);

int[] even =
    stream
        .filter(i -> i % 2 == 0)
        .toArray();

System.out.println(Arrays.toString(even));
```

```
[8, 12, 22, 2, 44]
```

In diesem Beispiel wird direkt ein Stream von `int`-Werten angelegt und mithilfe eines `IntPredicate i -> i % 2 == 0` gefiltert. Das Ergebnis wird in ein `int`-Array konvertiert und ausgegeben. Durch die Verwendung eines `IntStreams` kommt das Programm ohne jedes Boxing aus.

3.13 Beispiel: Umwandlung zu einem IntStream [37]

```
int summe =
    orders.stream()
        .flatMap(e -> e.getItems().stream())
        .mapToInt(Item::getPrice)
        .reduce((a, e) -> a + e)
        .orElse(0);

System.out.println(summe);
```

```
11910
```

Hier wurde noch einmal das vorhergehende Beispiel aufgegriffen und insofern optimiert, als dass bei der Map-Operation direkt über `mapToInt` in einen `IntStream` gemappt wird. Hierdurch ist beim `reduce` kein Boxing mehr nötig und das Programm läuft schneller und mit deutlich weniger Speicherverbrauch.

Kapitel 4

Fortgeschrittene Techniken

4.1 Reihenfolge der Elemente im Stream [39]

- Die Reihenfolge der Elemente im Stream nennt man **encounter order** — encounter order
  - Bei *geordneten Collections* übernimmt der Stream die Reihenfolge der Elemente aus der Collection (hat eine *definierte encounter order*)
  - Bei *ungeordneten Collections* haben die Elemente im Stream eine beliebige Reihenfolge (*encounter order ist undefiniert*)
- Ist die encounter order definiert, haben die erzeugten Streams diese Ordnung
- Die Ordnung kann über die `unordered` Methode abgeschaltet werden — unordered

Im Kapitel zu Collections wurde das Thema der Ordnung bereits behandelt. Während `List` eine Ordnung hält, ist das `Set` prinzipiell ungeordnet (mit Ausnahme des `TreeSet`, das ein `SortedSet` ist).

Dieselbe Überlegung muss man für Streams anstellen. Man kann zwar keine Elemente direkt im Stream ansprechen (daher der Name), man kann aber die Reihenfolge, mit der man die Elemente beim Durchlaufen des Streams antrifft (engl. encounter) notieren. Insofern kann man sich fragen, ob die *encounter order* eines Streams festgelegt (engl. defined) ist oder nicht.

Wenn man einen Stream aus einer Collection erzeugt, die geordnet ist, dann ist die encounter order des Streams definiert und entspricht der Ordnung der Collection. Diese wird über unterschiedliche Stream-Operationen erhalten und an erzeugte „Substreams“ weitergegeben – es sei denn, man ruft `unordered()` auf, worauf hin, die erzeugten Streams keine definierte encounter order mehr haben.

Ist die Collection aus der man den Stream erzeugt, selbst nicht geordnet, dann hat der Stream keine definierte encounter order. Es ist dann nicht vorhersagbar, welche Reihenfolge die Elemente eines „Substreams“ haben werden.

4.2 Collector [40]

Typ der von der `collect`-Operation erzeugten Collection kann der `toCollection()`-Methode mitgegeben werden — collect



```
Stream<String> stream = Stream.of("A", "B", "C");
List<String> list = stream.collect(toCollection(LinkedList::new));
```

Daten eines Streams können über Collector in einen einzigen Wert verdichtet werden

- Durchschnitt: `averagingDouble`, `averagingInt`, `averagingLong`
- Anzahl der Elemente: `counting`
- Maximum und Minimum: `maxBy`, `minBy`
- Summieren: `summingInt`, `summingDouble`, `summingLong`

Mittelwert berechnen

```
Stream<Double> stream = Stream.of(3.0, 2.0, 4.0, 1.0, 2.0, 2.7);
double average = stream.collect(averagingDouble(e -> e));
System.out.printf("%.2f%n", average); // -> 2,45
```

Summieren

```
Stream<Double> stream = Stream.of(3.0, 2.0, 4.0, 1.0, 2.0, 2.7);
double sum = stream.collect(summingDouble(e -> e));
System.out.printf("%.2f%n", sum); // -> 14,70
```

4.3 Daten aufteilen [43]

Über den Collector `partitioningBy` und ein `Predicate` kann man einen Stream aufteilen. Das Ergebnis ist eine `Map` mit den beiden Schlüsseln `Boolean.TRUE` und `Boolean.FALSE`.

partitioningBy
Predicate

```
Stream<Character> stream = Stream.of('A', '2', 'X', '!', '3', '7');

Map<Boolean, List<Character>> result =
       stream.collect(partitioningBy(Character::isLetter));

List<Character> letters = result.get(true);
List<Character> other = result.get(false);

System.out.println(letters); // -> [A, X]
System.out.println(other); // -> [2, !, 3, 7]
```

Der Einrag in der Map für den Schlüssel `true` enthält alle Elemente, für die das `Predicate` `true` ergeben hat (hier `Character::isLetter`), also im Beispiel alle Buchstaben. Unter dem Schlüssel `false` sind die Elemente zu finden, für die das Ergebnis `false` war.

4.4 Gruppieren von Daten [44]

Mit dem Collector `groupingBy` kann man Daten nach beliebigen Kriterien gruppieren. Hierzu muss eine Funktion übergeben werden, die das Gruppierungskriterium zurückgibt

groupingBy

```
Stream<String> stream = Stream.of("USW", "GGT", "UA", "ZB", "FKK",
       "MFG", "SOSE", "A", "H", "MM", "MBIT");

Map<Integer, List<String>> result =
       stream.collect(groupingBy(String::length));

System.out.println(result);
```

Ausgabe

```
{1=[A, H], 2=[UA, ZB, MM], 3=[USW, GGT, FKK, MFG], 4=[SOSE, MBIT]}
```

In diesem Beispiel werden die Strings in dem Stream nach ihrer Länge gruppiert. Die entsprechende Funktion zur Klassifizierung ist daher `String::length`. Der von der Funktion zurückgegebene Wert dient dann als Schlüssel in der Map, die die gruppierten Werte enthält.

4.5 Erzeugung von Strings [45]

Soll aus einem Stream ein String erzeugt werden, kann man den `joining` Collector verwenden

joining

```
Stream<String> stream = Stream.of("USW", "GGT", "UA", "ZB", "FKK",
       "MFG", "SOSE", "A", "H", "MM", "MBIT");

String result = stream.collect(joining(" | ", "{ ", " }"));

System.out.println(result);
```

Ausgabe

```
{ USW | GGT | UA | ZB | FKK | MFG | SOSE | A | H | MM | MBIT }
```

Man muss dem `joining`-Collector drei Informationen mitgeben:

- Trennzeichen (`delimiter`)
- Anfang (`prefix`)
- Ende (`suffix`)

4.6 Parallelisierung [46]

- Stream-Operationen können automatisch parallelisiert werden
- Anstatt `stream()` ruft man die Methode `parallelStream()` `parallelStream()`
- Java VM kümmert sich um die Parallelisierung der Operationen
- Lohnt sich erst ab einer gewissen Größe der Streams
- Für den/die Programmierer:in ändert sich sonst nichts

In einem anderen Kapitel gehen wir noch auf das Thema Thread-Programmierung ein. Dort werden ausgiebig die Probleme behandelt, die durch die Parallelität entstehen und wie man ihnen begegnen kann.

Der große Vorteil des funktionalen Paradigmas, wie es bei den Streams eingesetzt wird, ist, dass man es automatisch parallelisieren kann, ohne dass Entwickler:innen etwas tun müssen. Der Grund liegt darin, dass die verwendeten Lambda-Funktionen **seiteneffektfrei** sind und damit kein gleichzeitiger Zugriff auf Daten durch die Funktionen erfolgen kann. seiteneffektfrei

Details zu diesem Thema würden den Rahmen dieses Kurses sprengen. Es sei aber angemerkt, dass die funktionale Programmierung deutlich weniger Probleme bei der Parallelisierung erzeugt als die klassische imperative Programmierung und daher im Augenblick eine Renaissance erlebt.

Index

autoboxing, 17
autounboxing, 17

boxing, 17

eager, 5
encounter order, 19
externe Iteration, 2

intermediate operations, 5
interne Iteration, 3

lazy, 5

method chaining, 6

seiteneffektfrei, 22
Streams, 3

terminal operation, 7
terminal operations, 5

unboxing, 17

Wrapper-Objekt, 17

Programmieren 2 (PR2)

Vorlesung - Technische Hochschule Mannheim

Threads



Wintersemester 2025/2026

Prof. Thomas Smits

Diese Materialien sind ausschließlich für den persönlichen Gebrauch durch Besucher der Vorlesung Programmieren 2 (PR2) an der Technischen Hochschule Mannheim gedacht. Jede andere Nutzung ist ohne vorherige Zustimmung des Autors nicht zulässig.

Stand 2025-09-13

Technische Hochschule
mannheim

Inhaltsverzeichnis

1 Begriffe und Leitfragen **1**

1.1 Fragen [5] . . . 1
1.2 Begriffe [6] . . . 1
1.3 Links zur Java-Insel [7] . . . 1

2 Motivation **2**

2.1 Wozu braucht man Nebenläufigkeit? [9] . . . 2
2.2 Moore's Law [10] . . . 3
2.3 Prozessor Taktraten: 2002 [11] . . . 4
2.4 Prozessor Taktraten: 2007 [12] . . . 4
2.5 Nebenläufigkeit [13] . . . 5
2.6 Thread versus Prozess [14] . . . 5
2.7 Thread versus Prozess: Prozesse [16] . . . 6
2.8 Thread versus Prozess: Threads [17] . . . 6

3 Threads in Java – Grundlagen **7**

3.1 Gründe für den Einsatz von Threads [19] . . . 7
3.2 Threads in Java [20] . . . 7
3.3 Beispiel: Thread (preemtiv) [21] . . . 8
3.4 Beispiel: Thread (kooperativ) [25] . . . 10
3.5 Beispiel: Thread mit Namen [27] . . . 11
3.6 Beispiel: Stack versus Heap 1 [30] . . . 12
3.7 Beispiel: Stack versus Heap 2 [33] . . . 13
3.8 Die Klasse Thread [36] . . . 14
3.9 Zustände eines Threads [38] . . . 14
3.10 Beispiel: Thread noch einmal starten [39] . . . 15
3.11 Beenden eines Threads [40] . . . 16
3.12 Beispiel: Thread beenden (Flag) [41] . . . 16
3.13 Beispiel: Thread beenden (interrupt()) [43] . . . 17
3.14 Threads vereinigen [47] . . . 19
3.15 Beispiel: Threads vereinigen [48] . . . 19
3.16 Prioritäten und Daemon-Threads [51] . . . 20
3.17 Beispiel: Daemon-Threads [52] . . . 21

4 Schrecken der Nebenläufigkeit 22
4.1 Das Therac-25 Disaster [55] . . . 22
4.2 Probleme der Thread-Programmierung [56] . . . 23
4.3 Prozessor [57] . . . 24
4.4 Multicore und Caches [58] . . . 25
4.5 NUMA - Architekturen [59] . . . 25
4.6 Problemklassen [60] . . . 26
4.7 Beispiel: Safety Hazard (race condition) [61] . . . 26
4.8 Beispiel: Safety Hazard (check-then-act) [63] . . . 27
4.9 Beispiel: Liveness Hazard (Livelock) [64] . . . 28
4.10 Threadsafety [65] . . . 29
4.11 Wichtige Fragen bei Threads [66] . . . 29
4.12 Atomare Operationen [67] . . . 29
4.13 Ordering in Java [68] . . . 30
4.14 Was kommt hier raus? [69] . . . 31
4.15 Das Java Memory Modell [71] . . . 31
4.16 Garantien des Memory-Modells [72] . . . 32
4.17 Das Schlüsselwort volatile [73] . . . 33
4.18 Beispiel: volatile [74] . . . 33
4.19 Das Schlüsselwort synchronized [75] . . . 34
4.20 Beispiel: Ungeschützte Klasse [76] . . . 35
4.21 Beispiel: Synchronized bei Methoden [77] . . . 35
4.22 Beispiel: Synchronized Blöcke [79] . . . 37
4.23 Wie funktioniert synchronized? [80] . . . 37
4.24 Wie funktionieren synchronized genau? [81] . . . 38
4.25 Synchronized im Zustandsdiagramm [82] . . . 38
4.26 Java Locks sind reentrant [83] . . . 38
4.27 Synchronized und das Memory Model [85] . . . 39
4.28 Synchronized und Leseoperationen [86] . . . 39
4.29 Lifelock und Synchonized [87] . . . 40
4.30 Deadlocks [90] . . . 42
4.31 Beispiel: Deadlock [92] . . . 42
4.32 Tipp: Dedizierte Lock-Klassen [94] . . . 43
4.33 Thread Local Storage [96] . . . 44
4.34 Beispiel: ThreadLocal [97] . . . 44
4.35 Amdahl's Law [99] . . . 45

5 Thread-Koordination 47
5.1 Das Problem des schlafenden Friseurs [102] . . . 47
5.2 Das Problem [103] . . . 47
5.3 Die Lösung: wait() und notify() [104] . . . 48
5.4 Zustandsdiagramm mit Wait-Pool [105] . . . 48
5.5 Beispiel: Producer / Consumer [106] . . . 49
5.6 Beispiel: Producer – run()-Methode [107] . . . 49
5.7 Beispiel: Consumer – run()-Methode [108] . . . 50

5.8 Beispiel: Start-Code [109] . . . 50
5.9 Wie funktionieren wait() und notify()? [111] . . . 51
5.10 Was muss ich mir merken… [112] . . . 51

6 java.util.concurrent 52
6.1 Timer und TimerTask [114] . . . 52
6.2 Beispiel: Timer / TimerTask [115] . . . 52
6.3 java.util.concurrent [118] . . . 53
6.4 Producer/Consumer mit BlockingQueue [119] . . . 54
6.5 Beispiel: BlockingQueue (Producer) [120] . . . 54
6.6 Beispiel: BlockingQueue (Consumer) [121] . . . 54
6.7 CountDownLatch [122] . . . 55
6.8 Beispiel: Threads vereinigen mit Latch [123] . . . 55
6.9 Executor [126] . . . 56
6.10 Beispiel: Executor [127] . . . 56
6.11 Future [129] . . . 57
6.12 Beispiel: Callable [130] . . . 57
6.13 Beispiel: Future [132] . . . 58
6.14 Atomics [133] . . . 59

Index iii

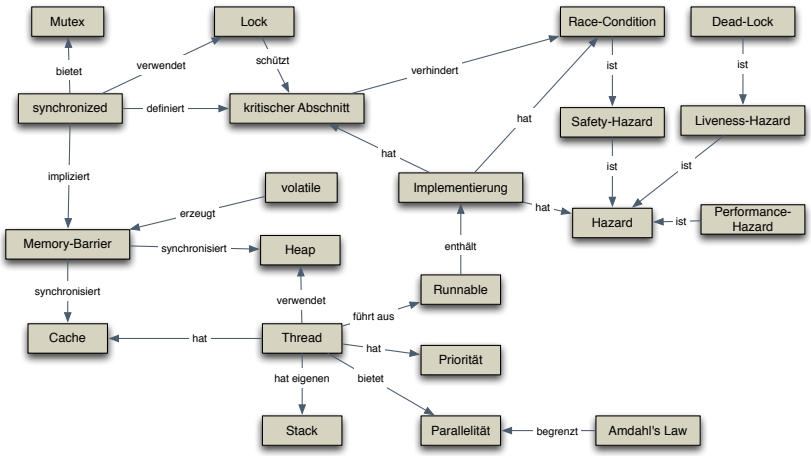
Kapitel 1

Begriffe und Leitfragen

1.1 Fragen [5]

- Wie kann ich die vielen Prozessorkerne ausnutzen?
- Wie kann ich in Java einfach parallele Programme schreiben?
- Welche Probleme tauchen bei der Thread-Programmierung plötzlich auf?
- Wie kann ich sicher mit Threads umgehen?

1.2 Begriffe [6]



Alle Java-Quellen zu diesem Kapitel finden sich auf Gitty.

1.3 Links zur Java-Insel [7]

- 17. Einführung in die nebenläufige Programmierung

Kapitel 2

Motivation

2.1 Wozu braucht man Nebenläufigkeit? [9]

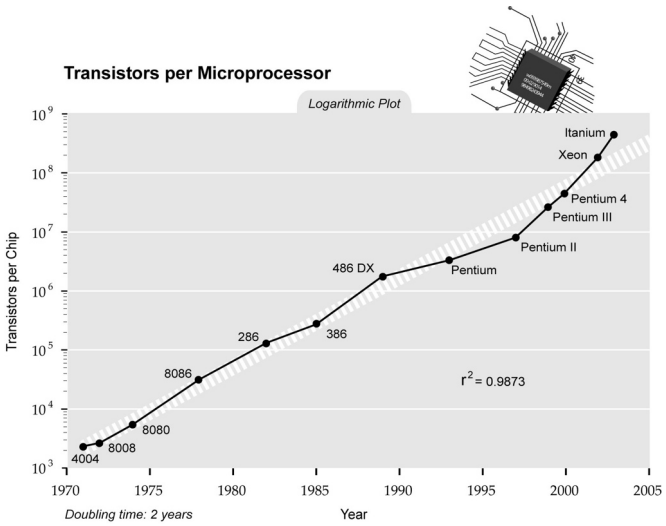
- Ein Webserver soll mehrere Verbindungen gleichzeitig bearbeiten können
- Ein User-Interface soll nicht blockieren, obwohl eine Berechnung im Hintergrund läuft
- Eine Rechtschreibprüfung soll neben dem normalen Editieren des Dokuments ablaufen
- Eine App soll eine Animation anzeigen und gleichzeitig Daten aus dem Netzwerk nachladen
- Ein mathematisches Problem soll auf mehrere CPUs verteilt werden

Viele Probleme kann man nur lösen, wenn man innerhalb eines Programms mehrere parallele oder quasi-parallele Aktionen zulässt. So muss z. B. ein Webserver in der Lage sein, mehrere Benutzeranfragen (requests) parallel zu verarbeitet. Würde man hier keine Parallelität zulassen, könnte er nur eine Anfrage nach der anderen beantworten, was für die Benutzer:innen nicht akzeptabel wäre.

Genauso gibt es viele andere Dinge, die man gerne parallel ausführen möchte, um das Benutzer-Interface der Anwendung ansprechender zu gestalten. Beispiele sind Animationen oder eine Rechtschreibprüfung, die während der Eingabe direkt die Wörter überprüft. Damit hier der/die Benutzer:in durch die Prüfung in seinem Arbeitsfluss nicht beschränkt wird, muss diese parallel laufen.

Ein weiterer Grund für den Einsatz paralleler Programmierung sind Probleme, die auf mehrere CPUs und CPU-Kerne verteilt werden sollen. Moderne Supercomputer bestehen aus tausenden von CPUs und nur die Parallelisierung der Programme erlaubt es deren Rechenleistung auszunutzen. Da Multiprozessorsysteme heute bereits bei Heimcomputern zum Standard gehören, nimmt die Bedeutung der Parallelprogrammierung immer weiter zu.

2.2 Moore's Law [10]

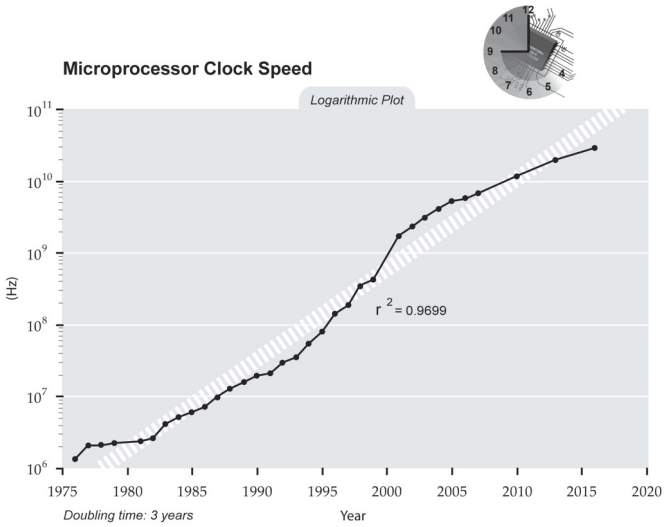


Moores Gesetz (Moore's law) wurde ursprünglich formuliert in „Cramming more components onto integrated circuits", *Electronics Magazine 19 April 1965* und lautet: Moores Gesetz

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year… Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.
>
> Moore, Cramming more components onto integrated circuits", Electronics Magazine 19 April 1965

Es handelt sich dabei natürlich nicht um ein Gesetzt im engeren Sinne, sondern eher um eine Prophezeiung. Interessanterweise ist diese aber bis heute, also nach fast 50 Jahren, immer noch korrekt.

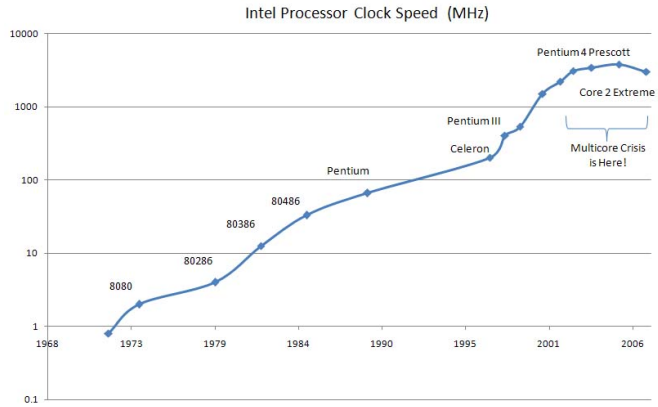
2.3 Prozessor Taktraten: 2002 [11]



Die zusätzlichen Transistoren, die in den Schaltkreisen zur Verfügung standen, wurden primär dazu genutzt, die Taktraten der Prozessoren zu steigern.

Diese Grafik zeigt eine Vorhersage von Ray Kurzweil aus dem Jahre 2005. Kurzweil hat damals noch geglaubt, dass die Taktraten weiterhin parallel mit der Anzahl der Transistoren steigen. Diese Vorhersage hat sich nicht bewahrheitet, wie wir heute wissen, da durch die Taktraten der Energieverbrauch der Prozessoren ebenfalls steigt und Bereiche erklommen hat, die als nicht mehr akzeptabel gelten. Insofern haben wir im Jahre 2010 nicht die 10 GHz-Prozessoren gesehen, die Kurzweil 2005 vorhergesagt hat.

2.4 Prozessor Taktraten: 2007 [12]



Die tatsächlichen Taktraten der Prozessoren stagnieren seit Mitte der 2000 Jahre. Die zusätzlichen Transistoren fließen daher nicht mehr in höhere Takte, sondern in mehr Prozessorkerne

innerhalb eines Chips. Ein moderner Prozessor enthält also auf einem Chip mehrere vollkommen unabhängige Recheneinheiten und besteht eigentlich aus vielen Prozessoren; diese werden als **Kerne** (cores) bezeichnet. Deshalb spricht man auch von **Multi-Core-Prozessoren**.

Kerne
Multi-Core-Prozessoren

2.5 Nebenläufigkeit [13]

Nebenläufigkeit ist die Fähigkeit eines Systems mehrere Aktionen parallel oder quasi-parallel durchzuführen

- Man unterscheidet zwischen *Threads* (Fäden) und *Prozessen*
- Wichtige Begriffe
  - Prozesse
  - Native Threads, Betriebssystem-Threads
  - Green-Threads, User-Level Threads
  - Fibers, Co-Routinen
- Scheduling und Scheduler (preemtiv, kooperativ)

Ein **Scheduler** (Scheduler = Steuerprogramm; vom englischen schedule für „Zeitplan“) ist eine Arbitrationslogik, die die zeitliche Ausführung mehrerer Prozesse (oder Threads) in Betriebssystemen regelt. Scheduler kann man grob in **unterbrechende** (präemptiv) und **nicht unterbrechende** (non preemptive) (manchmal **kooperativ** genannt) aufteilen. Nicht unterbrechende Scheduler lassen einen Prozess, nachdem ihm die CPU einmal zugeteilt wurde, solange laufen, bis dieser diese von sich aus wieder freigibt oder bis er blockiert. Unterbrechende Scheduler teilen die CPU von vornherein nur für eine bestimmte Zeitspanne zu und entziehen dem Prozess diese daraufhin wieder. Quelle: Wikipedia

Scheduler
unterbrechende
nicht unterbrechende
kooperativ

2.6 Thread versus Prozess [14]

- **Prozesse**
  - getrennte Adressräume (getrennter Heap)
  - Kommunikation nur über Inter Process Communication (IPC)
  - kann einen oder mehrere Threads enthalten
  - schwergewichtig
  - eigene Ressourcen (geöffnete Dateien, Sockets, Speicher etc.)
- **Threads**
  - gemeinsamer Adressraum (gemeinsamer Heap)
  - getrennte Stacks
  - Kommunikation über den gemeinsamen Speicher
  - leichtgewichtiger als Prozesse

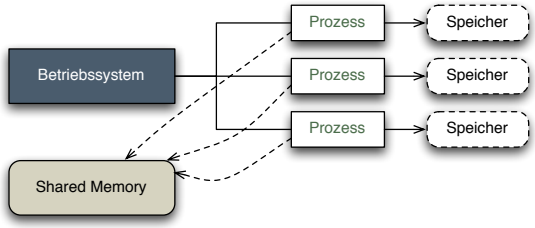
Prozesse
Threads

Programme werden vom Betriebssystem als *Prozesse* ausgeführt. Prozesse sind aus Sicherheitsgründen vollständig voneinander isoliert und haben eigene Ressourcen. Vor allem ist der Speicher von Prozessen vollständig getrennt, d. h. ein Prozess kann nicht auf den Speicher eines anderen Prozesses zugreifen. Prozesse können daher nur über den Umweg des Betriebssystems miteinander in Kontakt treten. Die Technik hierzu nennt man **Interprozess-Kommunikation** (interprocess communication) **IPC**. Beispiele für IPC sind Sockets, Pipes und Shared Memory. Wegen der vollständigen Isolation sind Prozesse relativ schwergewichtig, d. h. das Starten und die Verwaltung von Prozessen benötigt relativ viele Ressourcen.

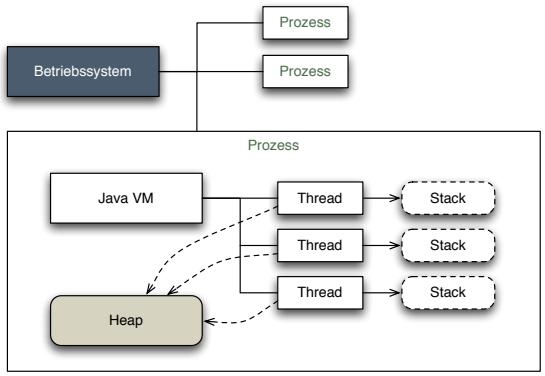
Interprozess-Kommunikation
IPC

Innerhalb eines Prozesses gibt es einen oder mehrere *Threads*, die man sich als Ausführungspfade vorstellen kann. Im Gegensatz zu Prozessen sind Threads leichtgewichtiger, d. h. sie lassen sich mit deutlich weniger Aufwand starten und verwalten. Da Threads innerhalb eines Prozesses laufen, haben sie keinen getrennten Heap, sondern teilen sich den Heap und kommunizieren über diesen miteinander. Der Stack ist aber bei Threads getrennt: Jeder Thread hat seinen eigenen Stack.

2.7 Thread versus Prozess: Prozesse [16]



2.8 Thread versus Prozess: Threads [17]



Kapitel 3

Threads in Java – Grundlagen

3.1 Gründe für den Einsatz von Threads [19]

- Nichtblockierende Eingabe/Ausgabe (nonblocking I/O)
- Zeitgesteuerte Aufgaben (timer, alarms)
- Unabhängige Aufgaben (independent tasks)
- Parallelisierbare Algorithmen (paralelizable algorithms)

3.2 Threads in Java [20]

- Threads sind in Java (wie in ADA) ein Teil der Sprache und nicht, wie in C oder C++, Bibliotheksfunktionen
- Die Sprachspezifikation macht keine Aussage zur Implementierung (*native* / *preemtive* vs. *green* / *kooperativ*)
  - Alle gängigen Java-Implementierungen (Oracle, Apple, IBM, SAP, HP) bieten native Threads mit preemtivem Scheduling
- Basis aller Thread-Programmierung in Java sind die Klasse `Thread` und das Interface `Runnable`
- Seit Java 1.5 gibt es mit `java.util.concurrent` neue Möglichkeiten der parallelen Programmierung – `Thread` bleibt aber die Grundlage
  - `java.util.concurrent` ist einer direkten Verwendung von `Thread` vorzuziehen

Als man die Sprache Java entworfen hat, hat man entschieden, dass viele komplizierte Techniken von C/C++ nicht Einzug in Java halten sollten. Deshalb wurden die Eingabe-/Ausgabe-Operationen vollständig synchron entworfen. Dies hat aber zur Folge, dass man sofort ein Konzept zur Nebenläufigkeit mit Threads benötigt, da man andernfalls keine ernsthaften Programme schreiben könnte. Aus diesem Grund wurden Threads Teil der Sprachspezifikation und sind daher auf jeder Java-Implementierung und Plattform verfügbar.

Es gibt zwei Möglichkeiten Threads zu implementieren

1. **preemptives Scheduling** (auch **native threads** genannt): Die Verwaltung der Threads wird vom Betriebssystem übernommen. Die **Zeitscheiben** (wie lange ein Thread lau-

preemptives Scheduling
native threads
Zeitscheiben

fen darf) werden hart durchgesetzt. Wenn ein Thread seine Zeitscheibe aufgebraucht hat, wird er (auch gegen seinen Willen) unterbrochen und ein anderer Thread darf weiterarbeiten.

2. **kooperatives Scheduling** (auch **green threading** genannt): Die Threads werden von der Virtuellen Maschine selbst verwaltet. Die Threads geben sich gegenseitig Rechenzeit ab. Ein Thread läuft solange bis er selbst der VM mitteilt, dass ein anderer Thread laufen soll.

kooperatives Scheduling

green threading

Da das kooperative Modell deutlich fehleranfälliger ist, verwenden die allermeisten Virtuellen Maschinen native Threads.

Das zentrale Element der Thread-Programmierung in Java sind die Klasse `Thread` und das Interface `Runnable`. Der Umgang mit diesen beiden wird uns in den nächsten Abschnitte beschäftigen.

3.3 Beispiel: Thread (preemtiv) [21]

```
class Runner implements Runnable {
    public void run() {
        while (true) {
            System.out.println("Paralleler Thread");
        }
    }
}
```

Das Beispiel zeigt eine Klasse `Runner`, die das `Runnable`-Interface implementiert. Der Code, der in der `run`-Methode enthalten ist, wird später parallel zum Hauptprogramm ausgeführt werden.

```
public class ThreadDemo {
    public static void main(String[] args) throws Exception {

        Thread t = new Thread(new Runner());
        t.start();

        while (true) {
            System.out.println("Hauptprogramm");
        }
    }
}
```

```
Paralleler Thread
Hauptprogramm
Hauptprogramm
Paralleler Thread
```

```
Paralleler Thread
```

Von der Klasse `Runner` muss eine Instanz erzeugt werden, damit die nicht-statische `run`-Methode ausgeführt werden kann. Dies ist nötig, da man in Java keine Methoden übergeben kann, sondern nur Objekte. Das Objekt von `Runner` ist also nichts anderes als ein Hilfsmittel, um die `run`-Methode übergeben zu können.

`Runner`

Diese Instanz von `Runnable` kann dann einem Objekt der Klasse `Thread` im Konstruktor übergeben werden. Mit der Erzeugung des `Thread`-Objektes beginnt aber der neue Thread noch nicht zu laufen; dies passiert erst nachdem die `start()`-Methode gerufen wurde.

`Runnable`
`Thread`
`start()`

An der Ausgabe erkennt man deutlich, dass beide `while`-Schleifen parallel ausgeführt werden und beide Endlosschleifen Ausgaben erzeugen.

Wichtig ist, dass man folgenden Unterschied klar sieht:

1. Das `Runnable`-Objekt und dessen `run()`-Methode repräsentiert den Code, der parallel ausgeführt werden soll.
2. Die Klasse `Thread` führt dann den Code der `run()`-Methode aus.

Man kann sich `Thread` als einen „virtuellen“ Prozessor vorstellen, der dazu dient die parallelen Aufgaben auszuführen.

```java
public class ThreadDemo {

    public static void main(String[] args) throws Exception {

        Thread t = new Thread(
          () -> {
                while (true) {
                    System.out.println("Paraller Thread");
                }
        });
        t.start();

        while (true) {
            System.out.println("Hauptprogramm");
        }
    }
}
```

Dieses Beispiel ist äquivalent zum vorhergehenden, nur dass hier ein Lambda für die Erzeugung des `Runnable`-Objekts verwendet wird.

```
Paralleler Thread
Hauptprogramm
Hauptprogramm
Hauptprogramm
```

```
Paralleler Thread
Paralleler Thread
```

3.4 Beispiel: Thread (kooperativ) [25]

```
class Runner implements Runnable {
    public void run() {
        while (true) {
            System.out.println("Paralleler Thread");
            Thread.yield();
        }
    }
}
```



Der Unterschied zur preemptiven Variante ist, dass hier der Thread freiwillig Rechenzeit durch die Methode `Thread.yield()` abgibt. Durch den Aufruf von `yield()` teilt der Thread mit, dass er Rechenzeit abgeben möchte. Er bleibt dann einfach in der Methode stehen und einem anderen Thread wird vom Scheduler mit Rechenzeit versorgt. Es kann vorkommen, dass kein anderer Thread Rechenzeit angefordert hat; in diesem Fall kehrt der Aufruf von `yield()` sofort zurück und der Thread läuft einfach weiter.

yield()

`yield()` aufzurufen garantiert also nicht, dass wirklich ein anderer Thread an die Reihe kommt. Die letztendliche Entscheidung liegt beim Scheduler. Der `yield()`-Aufruf ist somit nur ein Hinweis an den Scheduler, bietet aber keine Garantien.

```
public class ThreadDemo {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new Runner());
        t.start();

        while (true) {
            System.out.println("Hauptprogramm");
            Thread.yield();
        }
    }
}
```



```
Paralleler Thread
Hauptprogramm
Paralleler Thread
Hauptprogramm
Paralleler Thread
```



Dass die Ausgaben hier genau abwechselnd erfolgen ist Zufall. Wegen des oben bereits beschriebenen Verhalten von `yield()` kann es auch vorkommen, dass viele Male hintereinander eine der beiden Threads Ausgaben machen kann.

3.5 Beispiel: Thread mit Namen [27]

```
class Runner implements Runnable {
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName());
        }
    }
}
```

Man kann einem Thread beim Erzeugen einen Namen mitgeben. Diesen kann dann die ausgeführte Methode über `Thread.currentThread().getName()` wieder auslesen.

```
public class ThreadDemo {
    public static void main(String[] args) throws Exception {

        Runnable runner = new Runner();

        Thread t1 = new Thread(runner, "Thread-1");
        Thread t2 = new Thread(runner, "Thread-2");
        t1.start(); t2.start();

        while (true) {
            System.out.println("Hauptprogramm");
        }
    }
}
```

Man kann einem Thread bei der Erzeugung über den Konstruktor einen Namen mitgeben. Dieser kann aus dem Thread über `getName()` wieder ausgelesen werden. Die `run()`-Methode kann herausfinden, welcher Thread sie gerade ausführt, indem sie die statische Methode `Thread.currentThread()` verwendet.

`getName()`

In diesem Beispiel ist die Besonderheit, dass es nur ein Objekt von der Klasse `Runner` (und damit nur eine `run()`-Methode) gibt, diese aber von zwei Threads ausgeführt wird.

Dies macht noch einmal deutlich, dass man streng zwischen dem Thread als dem Ausführenden und der `Runnable`-Instanz als dem Ausgeführten trennen muss.

```
Thread-1
Thread-2
```

```
Thread-1
Hauptprogramm
Thread-2
Hauptprogramm
Thread-2
```

3.6 Beispiel: Stack versus Heap 1 [30]

```
class Runner4 implements Runnable {

    private int zaehler = 0;

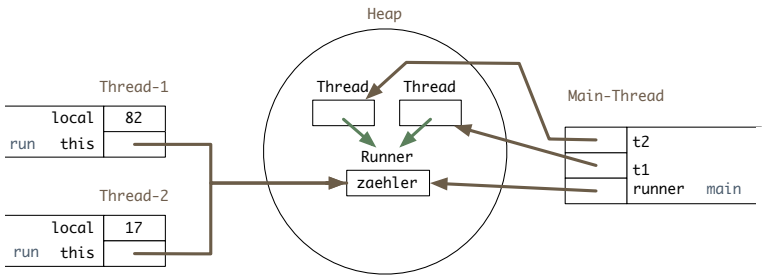
    public void run() {
        int local = new Random().nextInt();

        while (true) {
            zaehler = local++;
        }
    }
}
```

```
public class SimpleHeapAndStack {

    public static void main(String[] args) {
        Runner4 runner = new Runner4();
        Thread t1 = new Thread(runner, "Thread-1");
        Thread t2 = new Thread(runner, "Thread-2");

        t1.start();
        t2.start();
    }
}
```



Dieses Beispiel zeigt die Beziehung zwischen den verschiedenen Objekten. Auf dem Heap gibt es zwei Thread-Objekte, die beide eine `run()`-Methode ausführen. Da es drei Threads gibt (main-Thread und zwei explizit gestartete Threads) existieren drei Stacks.

3.7 Beispiel: Stack versus Heap 2 [33]

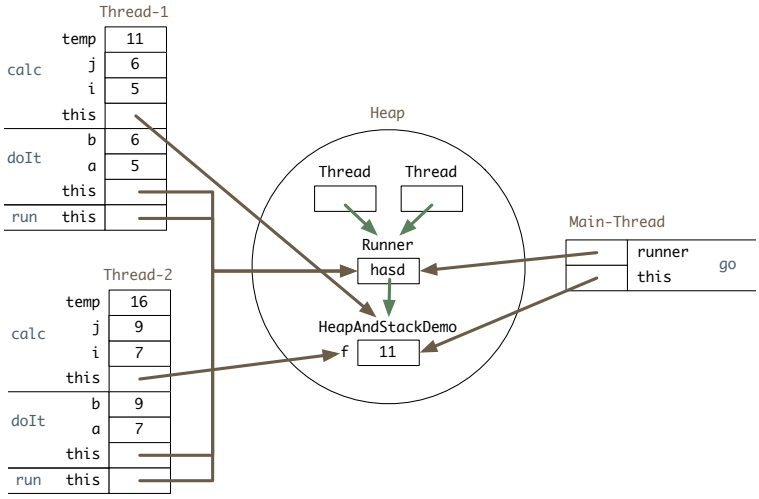
```
class Runner3 implements Runnable {
    private HeapAndStackDemo hasd;
    public Runner3(HeapAndStackDemo hasd) {
        this.hasd = hasd;
    }
    private void doIt() {
        int a = new Random().nextInt(10);
        int b = new Random().nextInt(10);
        hasd.f = hasd.calc(a, b);
    }
    public void run() {
        while (true) {
            doIt();
            System.out.println(Thread.currentThread().getName()
                    + ": " + hasd.f);
        }
    }
}
```

```
public class HeapAndStackDemo {

    int f;

    public int calc(int i, int j) {
        int temp = i + j;
        return temp;
    }

    public void go() {
        Runnable runner = new Runner3(this);
        new Thread(runner, "Thread-1").start();
        new Thread(runner, "Thread-2").start();
    }
}
```

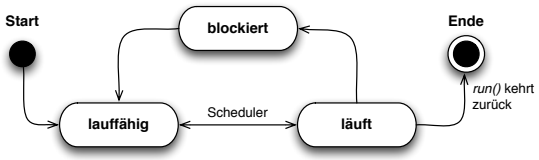


3.8 Die Klasse Thread [36]

Methoden von Thread

- `static void yield()` – Rechenzeit abgeben
- `static void sleep(long millis)` – Rechenzeit abgeben und für `millis` Millisekunden schlafen
- `static Thread currentThread()` – Liefert den aktuellen Thread zurück
- `Thread(Runnable target)` – Einen neuen Thread anlegen
- `Thread(Runnable target, String name)` – Einen neuen Thread mit Namen `name` anlegen
- `void setName(String name)`, `String getName()` – Namen setzen und lesen
- `void start()` – Thread starten
- `void run()` – Thread implementiert selbst Runnable, man kann also von Thread ableiten und dann `run()` überschreiben (nicht empfohlen)

3.9 Zustände eines Threads [38]



Ein Thread kann verschiedene Zustände einnehmen. Diese Abbildung zeigt drei (der insgesamt fünf) Zustände. Die noch fehlenden zwei Zustände werden später nachgereicht.

Wenn ein Thread gestartet wird, läuft er nicht sofort los, sondern befindet sich im Zustand **lauffähig** (runnable). Dieser Zustand bedeutet, dass der Thread prinzipiell alle Voraussetzungen erfüllt, um ausgeführt zu werden. Da aber nur der Scheduler die CPU an Threads zuteilen darf, muss der Thread solange im Zustand lauffähig warten, bis der Scheduler ihn zur Ausführung bringt.

lauffähig

Wenn der Thread ausgeführt wird, ist er im Zustand **läuft** (running). Hier bleibt er solange, bis eine der folgenden Dinge passiert:

läuft

- Die Zeitscheibe des Threads ist zu Ende und der Scheduler entzieht ihm die CPU. Dann wandert er in den Zustand lauffähig zurück.
- Der Thread gibt per `Thread.yield()` den Rest seiner Zeitscheibe ab und wird vom Scheduler in den Zustand lauffähig versetzt.
- Der Thread führt eine Aktion durch, bei der er nicht weiterlaufen kann, z. B. weil er auf Daten aus einer Datei wartet oder sich selbst mit `Thread.sleep()` für eine gewisse Zeit schlafen gelegt hat. In diesem Fall wandert er in den Zustand **blockiert** (blocked). In diesem verbleibt er solange, bis die blockierende Situation verschwunden ist, woraufhin er wieder lauffähig wird.

blockiert

3.10 Beispiel: Thread noch einmal starten [39]

```
class Runner2 implements Runnable {
    public void run() { /* ... */ }
}

public class BrokenRestart {
    public static void main(String[] args) throws Exception {
        Runnable runnable = new Runner2();
        Thread t1 = new Thread(runnable, "Thread-1");
        t1.start();
        Thread.sleep(100);
        t1.start(); // FEHLER!!
    }
}
```

```
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Thread.java:613)
    at de.smits_net.pr2.runnable.BrokenRestart.main(BrokenRestart.java:18)
```

Es ist nicht möglich einen bereits gestarteten Thread noch einmal zu starten. Wurde die `start()`-Methode schon aufgerufen, führt jeder weitere Aufruf von `start()` zu einer `IllegalThreadStateException`.

3.11 Beenden eines Threads [40]

- Ein Thread endet, wenn die `run()`-Methode zurückkehrt
- Man kann den Wunsch, dass der Thread sich beenden soll durch das Setzen einer Variable von außen signalisieren
- Auf *keinen Fall* sollte man die folgenden Methoden verwenden, um einen Thread zu beenden
  - `void stop()`, `void stop(Throwable obj)`
  - `void destroy()`
  - `void suspend()`, `void resume()`
- Eine weitere Möglichkeit besteht darin, dem Thread mit `interrupt()` zu beenden
  - Hat den Vorteil, dass hier der Thread auch beendet werden kann, wenn er in bestimmten Methoden blockiert ist

`stop()`
`stop(Throwable obj)`
`destroy()`
`suspend()`
`resume()`
`interrupt()`

Ein Thread ist beendet, wenn die `run()`-Methode zurückkehrt, z. B. mit `return`. Da nur der Thread seinen internen Zustand kennt, ist die einzig adäquate Methode ihm zu signalisieren, dass er sich bei der nächsten Möglichkeit beenden soll. Hierzu kann man entweder ein `boolean` Flag setzen oder die `interrupt()`-Methode von `Thread` verwenden (Details folgen).

Es gibt eine ganze Reihe von Methoden in der Klasse `Thread`, die dem Beenden und Anhalten von Threads dienen. Diese Methoden sind als *überholt* (deprecated) gekennzeichnet und sollten auf gar keinen Fall benutzt werden. Da sie den Thread einfach an einer beliebigen Stelle stoppen, können unvorhergesehene Probleme auftreten. Insbesondere Ressourcen, die der Thread benutzt (Dateien, Speicher, Netzwerk- oder Datenbankverbindungen) werden nicht mehr korrekt freigegeben, wenn der Thread einfach von außen beendet wird.

3.12 Beispiel: Thread beenden (Flag) [41]

```
public class Stoppable implements Runnable {
    private volatile boolean cont = true;

    public void requestTermination() {
        cont = false;
    }

    public void run() {
        while (cont) {
            System.out.println("Thread laeuft");
            Thread.yield();
        }
        System.out.println("** Stoppable gestoppt **");
    }
}
```

```
Stoppable st = new Stoppable();
Thread thread = new Thread(st);
thread.start();
Thread.sleep(100);
st.requestTermination();
```

In diesem Beispiel wird dem Thread durch die Variable `cont` signalisiert, wenn er sich beenden soll. Das `volatile` bei der Deklaration kann an dieser Stelle noch nicht begründet werden, wird aber später noch erklärt. Es ist nötig, damit das Beenden korrekt funktioniert.

volatile

Wenn der Verwender möchte, dass sich der Thread beendet, kann er die `requestTermination()`-Methode auf der `Runnable`-Instanz aufrufen. Die `run()`-Methode kann dann beim nächsten Durchgang die Variable `cont` abfragen und sich sauber beenden.

3.13 Beispiel: Thread beenden (interrupt()) [43]

```
public class Interruptible implements Runnable {

    public void run() {

        while (true) {
            System.out.println("Thread laeuft");

            if (Thread.currentThread().isInterrupted()) {
                break;
            }
            Thread.yield();
        }
        System.out.println("** Thread gestoppt **");
    }
}
```

```
Interruptible interuptible = new Interruptible();

Thread thread = new Thread(interuptible);

thread.start();

Thread.sleep(100);

thread.interrupt();
```

Eine andere Möglichkeit einem Thread zu signalisieren, dass er sich beenden soll, besteht in der Methode `interrupt()` der Klasse Thread. Der Name ist insofern irreführend, als dass

interrupt()

er den Eindruck erweckt, der Thread würde durch den Aufruf unterbrochen. Tatsächlich wird aber nur eine Variable im Thread gesetzt, die die `run()`-Methode über `isInterrupted()` abfragen kann. Der Thread behält also hier die volle Kontrolle über die Beendigung und kann das `interrupt()` auch ignorieren.

`isInterrupted()`

Der große Vorteil dieser Variante gegenüber dem Flag besteht darin, dass es eine Reihe von Methoden gibt, in denen der Thread blockieren kann (z. B. `sleep()`). Würde man nur ein Flag setzen, der Thread könnte es nicht abfragen, da er im Zustand „blockiert“ ist. `interrupt()` hat den Vorteil, dass es den Thread aus dem blockierten Zustand wecken kann. Dies geschieht, indem die blockierten Methoden durch eine `InterruptedException` vorzeitig zurückkehren. Tritt diese Ausnahme auf, weiß der Thread, dass er im blockierten Zustand ein `interrupt()` empfangen hat. Deshalb ist das normale Verhalten, aus dem `catch`-Block der `InterruptedException` die `run()`-Methode mit `return` zu verlassen bzw. die Endlosschleife des Threads über ein `break` zu beenden.

Die `InterruptedException` ist insofern außergewöhnlich, als dass sie keinen Fehler signalisiert, sondern dazu dient, das frühzeitige Zurückkehren einer blockierenden Methode anzuzeigen. Wenn man z. B. `sleep(10000)` aufruft, um für 10 Sekunden zu schlafen, dann zeigt die `InterruptedException` an, dass der Thread kürzer als 10 Sekunden geschlafen hat, weil die Methode wegen eines `interrupt()`-Aufrufs früher zurückgekehrt ist.

```
public class Interruptible2 implements Runnable {
    public void run() {

        while (true) {
            System.out.println("Thread laeuft");

            if (Thread.currentThread().isInterrupted()) {
                break;
            }
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                break;
            }
        }
        System.out.println("** Thread gestoppt **");
    }
}
```

Der Code der main()-Methode ist hier unverändert, d. h. genauso wie auf der letzten Folie dargestellt:

```
    public static void main(String[] args) throws Exception {
        Interruptible2 interuptible = new Interruptible2();
        Thread thread = new Thread(interuptible);
        thread.start();
        Thread.sleep(100);
```

```
        thread.interrupt();
    }
```

```
public class Interruptible3 implements Runnable {

    public void run() {

        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("Thread laeuft");

            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("** Thread gestoppt **");
    }
}
```

Auch hier ist die main-Methode unverändert zum vorhergehenden Beispiel.

3.14 Threads vereinigen [47]

- Häufig möchte man, dass ein Thread erst weiterläuft, wenn andere Threads zu Ende gelaufen sind
- Mit der Methode `join()` kann man darauf warten, dass ein Thread fertig ist

  join()

  - `void join(long millis)` – Warte maximal `millis` Millisekunden (0 = ewig)
  - `void join(long millis, int nanos)` – Wartet maximal `millis` Milli- und `nanos` Nanosekunden
  - `void join()` – Wartet ewig
- Alle `join()`-Methoden sind unterbrechbar (**interruptible**), d. h. sie kehren mit einer `InterruptedException` zurück, wenn der Thread der sie aufgerufen hat (nicht der auf den gewartet wird) interrupted wird

  interruptible

3.15 Beispiel: Threads vereinigen [48]

```
class RunnerPrinter implements Runnable {
    String text;
    RunnerPrinter(String text) {
        this.text = text;
    }
    public void run() {
```

```
        for (int i = 0; i < 5; i++) {
            System.out.println(text);
            Thread.yield();
        }
        System.out.println(text + " ist fertig.");
    }
}
```

```
Thread t1 = new Thread(new RunnerPrinter("Runner 1"));
Thread t2 = new Thread(new RunnerPrinter("Runner 2"));
Thread t3 = new Thread(new RunnerPrinter("Runner 3"));

t1.start(); t2.start(); t3.start();
t1.join(); t2.join(); t3.join();

System.out.println("Alle fertig");
```



```
Runner 1
Runner 3
Runner 1
Runner 3
Runner 2
Runner 1 ist fertig.
Runner 3 ist fertig.
Runner 2
Runner 2
Runner 2 ist fertig.
Alle fertig
```



3.16 Prioritäten und Daemon-Threads [51]

- Man kann Threads Prioritäten zwischen 1 und 10 zuweisen
  - `void setPriority(int newPriority)`
  - `int getPriority()`
- Thread-Prioritäten sind nicht portabel und nicht zuverlässig – die VM sieht sie maximal als Hinweis
- Mann kann festlegen, was beim Beenden des Main-Threads passieren soll
  - `void setDaemon(boolean on)`
  - `boolean isDaemon()`
- Die VM beendet sich dann, wenn alle Nicht-Daemon-Threads beendet wurden

3.17 Beispiel: Daemon-Threads [52]

```
class DaemonRunner implements Runnable {
    public void run() {
        while (true);
    }
}

public class DaemonDemo {

    public static void main(String[] args) {
        Thread t = new Thread(new DaemonRunner());
        t.setDaemon(true);
        t.start();
    }
}
```



```
ublic class DaemonDemo {

    public static void main(String[] args) {
        Thread t = new Thread(() -> { while (true); });

        t.setDaemon(true);
        t.start();
    }
}
```



Kapitel 4

Schrecken der Nebenläufigkeit

4.1 Das Therac-25 Disaster [55]

Falsche Threadprogrammierung

Bilanz: drei Patienten getötet, mindestens drei schwer verletzt



Therac-25 war ein Linearbeschleuniger zur Anwendung in der Strahlentherapie. Er wurde von 1982 bis 1985 in elf Exemplaren von der kanadischen Regierungsfirma Atomic Energy of Canada Limited (AECL) gebaut und in Kliniken in den USA und in Kanada installiert. Durch Softwarefehler und mangelnde Qualitätssicherung war ein schwerer Funktionsfehler möglich, der von Juni 1985 bis 1987 drei Patienten das Leben kostete und drei weitere schwer verletzte, bevor geeignete Gegenmaßnahmen ergriffen wurden.

Dies ist einer der folgenschwersten Fehler in der Geschichte der Softwareentwicklung und ein oft studiertes Lehrbeispiel für Anforderungen an Software in sicherheitsrelevanten Bereichen.

Therac-25 war ein Elektronen-Linearbeschleuniger. Als therapeutische Strahlung, vor allem für die Krebstherapie, konnte entweder der Elektronenstrahl direkt oder die durch ein zwischengeschaltetes Target aus Wolfram erzeugte Röntgenstrahlung der Energie 25 MeV verwendet werden. Im direkten Modus wurde eine wesentliche geringere Brillanz (Stärke des Elektronenstrahls) eingestellt als im Röntgenmodus.

Die Vorgänger, Therac-6 und Therac-20, mit 6 bzw. 20 MeV Photonenenergie, waren nicht computerisierte Konstruktionen, bei denen die Sicherheitsmaßnahmen durch mechanische

Verriegelung und die Überwachung der Systemfunktion durch analoge Messgeräte realisiert waren. Ein Computer des Typs PDP-11 und ein VT-100 Terminal wurden später allein zur Erleichterung der Bedienung hinzugefügt.

Die Neukonstruktion Therac-25 ersetzte diese durch Sensoren, deren Messwerte vom Computer ausgewertet wurden, und Aktoren, die unter Softwarekontrolle die verschiedenen Einstellungen ausführten. Ein Prototyp, noch ohne die Computersteuerung, wurde 1976 fertiggestellt, die erste Serienmaschine 1982. 1983 wurde eine Sicherheitsanalyse des Geräts durchgeführt, die das Vertrauen in die Überlegenheit der Softwarelösung ausdrückte, da Software keinem Verschleiß unterworfen sei.

Alle Vorfälle beruhten darauf, dass der Linearbeschleuniger mit der hohen Brillanz für den Röntgenmodus arbeitete, aber das Wolfram-Target nicht im Strahlengang war. Dies ist der gefährlichste mögliche Betriebszustand, der bei den Vorgängermodellen durch eine mechanische Verriegelung ausgeschlossen war. Die Strahlenbelastung in den sechs Fällen wurde nachträglich zu 40 bis 200 Gray abgeschätzt, eine normale Behandlung entspricht einer Dosis unter 2 Gray. Eine Strahlenbelastung des gesamten Körpers mit 10 Gray gilt als sicher tödlich, für lokalisierte Strahlenbelastungen liegen wenig Erfahrungswerte vor.

Der Computer des Therac-25 war zugleich für die Messwerterfassung und Steuerung des Geräts, als auch für die Benutzerinteraktion zuständig, durch Multitasking wurden beide Aufgaben quasi-gleichzeitig erledigt. Das Kernproblem dabei war die korrekte Synchronisation der beiden Prozesse. Unter gewissen Umständen konnte es passieren, dass nach einer Korrektur der Eingabedaten durch den Bediener vom Computer bei der Ansteuerung des Gerätes nur bei einem Teil der Daten die korrigierten Daten, bei dem anderen aber die alten Daten vor der Korrektur verwendet wurden.

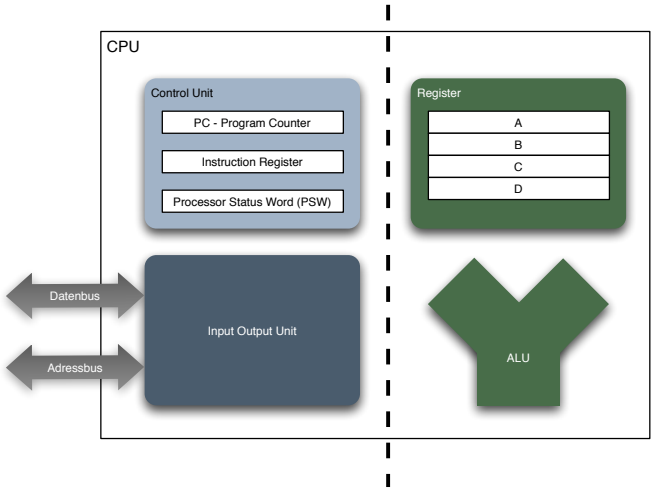
Quelle: Wikipedia

4.2 Probleme der Thread-Programmierung [56]

- Threads arbeiten im selben Adressraum, daher müssen Zugriffe auf Speicher koordiniert und geordnet erfolgen
- Thread kommunizieren über den gemeinsam genutzten Speicher, hier ist Koordination entscheidend
- Probleme im Bereich der Threads sind nur sehr schwer zu reproduzieren und treten häufig erst im Produktivsystem auf
- Parallelität und gemeinsame Speicher erfordern bestimme Sprachkonstrukte, die selbst zu neuen Problemen führen
- Menschen können nur schlecht in parallelen Abläufen denken

Threads lösen viele Probleme. Insbesondere erlauben sie, Programme deutlich schneller auszuführen. Mit Threads entstehen aber viele neue Probleme, da plötzlich parallele Aktionen auf den Speicher zugreifen. Ergreift man keine entsprechenden Maßnahmen kommt es zu Situationen, in denen nicht mehr vorhersehbar ist, was genau passiert und welchen Wert eine bestimmte Speicherstelle hat. Aus diesem Grund fließt ein großer Teil der Energie beim Programmieren mit Threads in die Schutzmaßnahmen.

4.3 Prozessor [57]



Moderne Prozessoren bestehen aus vier Grundeinheiten:

- Die **Control Unit** speichert Zustände des Prozessors und ist für das Laden und Abarbeiten der Befehle zuständig.
- Die **Input Output Unit** verbindet den Prozessor mit dem Speicher und weiteren Peripheriegeräten.
- Die **Arithmetisch-logische Einheit** (**ALU**) führt Rechenoperationen durch.
- Die **Register** bieten schnellen Speicher, auf dem die ALU arbeiten kann.

Control Unit

Input Output Unit

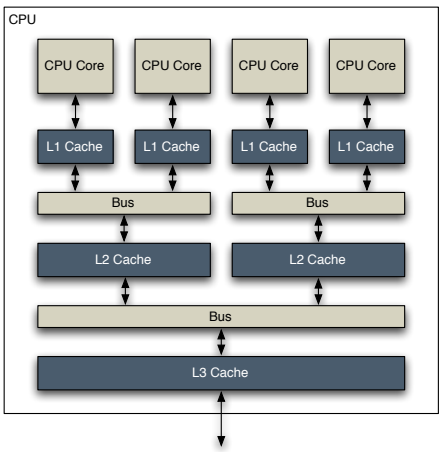
Arithmetisch-logische Einheit

ALU

Register

An dieser Stelle sind vor allem die Register für uns wichtig, da sie mit Daten aus dem Hauptspeicher befüllt werden, um dann Berechnungen anzustellen. Dies bedeutet insbesondere, dass der Hauptspeicher nicht immer eine aktuelle Sicht auf die Daten haben kann, da sich diese in den Register geändert haben können, aber noch nicht in den Hauptspeicher geschrieben wurden.

4.4 Multicore und Caches [58]

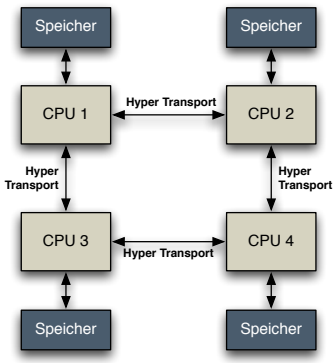


Die allermeisten der heute verfügbaren Prozessoren haben mehr als einen Kern und unterschiedliche Caches auf verschiedenen Ebenen. Somit ist die Frage „wo liegen die Daten?“ nicht mehr einfach zu beantworten, da Daten aus dem Hauptspeicher an unterschiedlichen Stellen abgelegt sein können:

- im Hauptspeicher selbst,
- in den Caches oder
- in den Registern des Prozessors.

An jeder dieser Stellen können die Daten in unterschiedlichen Versionen vorliegen.

4.5 NUMA - Architekturen [59]



NUMA steht für Non Uniform Memory Access NUMA

AMD Opteron benutzt schon seit längerem eine NUMA-Architektur mit Hyper Transport als Protokoll zwischen den Prozessoren. Intel folgt aber inzwischen ebenfalls diesem Paradigma

und integriert bei den aktuellen und zukünftigen Prozessoren ebenfalls den Speichercontroller in den Prozessor.

4.6 Problemklassen [60]

Safety Hazards

- **Safety Hazards** – Probleme bei denen sich das Programm in Anwesenheit mehrerer Threads nicht mehr korrekt verhält
  - race condition

Liveness Hazards

- **Liveness Hazards** – Probleme bei denen ein Programm bei mehreren Threads in einen Zustand gerät, bei dem es keine Fortschritte mehr machen kann
  - deadlock
  - starvation
  - livelock

Performance Hazards

- **Performance Hazards** – Probleme bei denen ein Programm zwar korrekt funktioniert, die Performance, trotz mehrerer Threads, jedoch schlecht ist

Die möglichen Probleme und Fehler, die mit Threads auftreten, kann man in drei Klassen einteilen. Während bei den *Safety Hazards* das Programm falsch arbeitet, bleibt es bei den *Liveness Hazards* einfach stehen. Nicht ganz so dramatisch sind die *Performance Hazards*, bei denen das Programm noch korrekt funktioniert aber nicht die gewünschte Performance zeigt.

Man unterscheidet innerhalb der einzelnen Fehlerklassen noch:

race condition

- **race condition**: Tritt auf, wenn sich zwei (oder mehrere) Threads ein Wettrennen liefern, z. B. beide versuchen gleichzeitig auf eine Variable zuzugreifen.

deadlock

- **deadlock**: Threads blockieren sich gegenseitig, keiner kann mehr weiterarbeiten, weil sie wechselseitig auf Ressourcen warten, die der jeweils andere belegt hat.

starvation

- **starvation**: Thread verhungert, weil er nicht mehr zu Zuge kommt, z. B. weil ein anderer Thread eine Ressource belegt und nicht mehr freigibt, die der Thread benötigt.

livelock

- **livelock**: Thread läuft in einer Schleife, deren Abbruchbedingung nie erfüllt wird.

4.7 Beispiel: Safety Hazard (race condition) [61]

```
public class SerialNumberGenerator {

    private int serienNummer = 0;

    public int naechsteNummer() {
        serienNummer++;
        return serienNummer;
    }
}
```

```
public class SerialNumberGenerator {

    private int serienNummer = 0;

    public int naechsteNummer() {

        int tmp = serienNummer;

        tmp = tmp + 1; // Thread 2

        serienNummer = tmp; // Thread 1

        return serienNummer;
    }
}
```

Das vorliegende Beispiel zeigt einen klassischen *Safety Hazards*. Die Operation `serienNummer++` sieht zwar aus, wie eine einzige Aktion, besteht aber in Wirklichkeit aus drei Schritten:

1. Der Wert der Variable muss aus dem Hauptspeicher in den Operanden-Stack der VM gelesen werden (hier symbolisiert durch die Variable `tmp`). Denn der Prozessor kann nicht direkt auf Hauptspeicher rechnen; die Daten müssen in ein Register geladen werden, damit die ALU damit umgehen kann.
2. Der Prozessor inkrementiert den Wert auf dem Operanden-Stack (im Register) mithilfe der ALU.
3. Der Wert der Variable wird zurück in den Hauptspeicher geschrieben.

Betrachtet man die drei Schritte, wird sofort klar, dass eine *race condition* vorliegt. Es kann sein, dass beide Threads, die gleichzeitig Variable aus dem Hauptspeicher auf den Stack kopieren und damit denselben Wert erhalten. Somit wird eine Seriennummer doppelt vergeben.

4.8 Beispiel: Safety Hazard (check-then-act) [63]

```
public class Singleton {

    private static Singleton instance;

    public static Singleton getInstance() {

        if (instance == null) { // Thread 2

            instance = new Singleton(); // Thread 1

        }

        return instance;
    }
```

```
}
```

Eine andere Art race condition liegt bei **check-then-act** vor. Hier wird eine Bedingung geprüft (check) und danach die Variable in der Bedingung verändert (act). Die Veränderung führt dann dazu, dass die Bedingung nicht mehr zutrifft. check-then-act

Diese Konstruktion ist besonders anfällig für race conditions, weil die Prüfung und die Veränderung zwei getrennte Operationen sind, die nacheinander erfolgen. Somit kann es passieren, dass zwei (oder mehrere) Threads die Bedingung prüfen *bevor* einer von ihnen die Bedingung ändern kann. Somit treten mehrere Threads in den Rumpf der Bedingung ein, obwohl die Aktion eigentlich nur einmal ausgeführt werden sollte.

Hierdurch werden im vorliegenden Beispiel mehr als ein Objekt vom Typ `Singleton` erzeugt.

4.9 Beispiel: Liveness Hazard (Livelock) [64]



```
class Looper implements Runnable {
    static boolean cont = true;

    public void run() {
        while (cont) {
            // mach was
        }
    }
}
public class Livelock {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new Looper());
        t.start();

        Thread.sleep(1000);
        Looper.cont = false;
    }
}
```

Das vorliegende Beispiel zeigt ein Livelock. Die Variable `cont` dient dazu, dem Thread, der `Looper` ausführt, zu signalisieren, dass er sich beenden soll. Die Veränderung erfolgt aus einem anderen Thread, dem Main-Thread.

Da die Variable nicht besonders gekennzeichnet ist (wir werden später noch `volatile` kennenlernen), wird es sehr häufig passieren, dass sie in einem Register oder einem Cache abgelegt wird. Da aber die Register pro Thread existieren können, wird die Änderung der Variable `cont` für den laufenden Thread gar nicht sichtbar, da der Main-Thread nur seine lokale Kopie der Variable verändert. Als Konsequenz läuft dann der Thread `Looper` ewig – es liegt ein Livelock vor.

4.10 Threadsafety [65]

threadsicher

- Eine Klasse ist **threadsicher** (threadsafe), wenn sie bei der Verwendung mehrerer Threads keine der genannten Probleme zeigt
- Viele Klassen der Klassenbibliothek sind nicht threadsafe (z. B. `SimpleDateFormat`) und müssen explizit geschützt werden
- Unveränderliche (immutable) Objekte sind immer threadsafe
- Die meisten Collections sind nicht threadsafe, können aber mit `Collections.synchronizedCollection()` entsprechend verpackt werden
- Von einigen Klassen gibt es zwei Varianten (`StringBuffer`, `StringBuilder`), die eine threadsafe, die andere nicht

4.11 Wichtige Fragen bei Threads [66]

Atomizität

Reihenfolge

Sichtbarkeit

- **Atomizität** (atomicity) – Welche Operationen eines Threads können von einem anderen unterbrochen werden, welche nicht?
- **Reihenfolge** (ordering) – In welcher Reihenfolge werden Operationen eines Threads ausgeführt?
- **Sichtbarkeit** (visibilitiy) – Wann werden Änderungen, die ein Thread macht, für andere Threads sichtbar?

Wenn man sich darüber Gedanken macht, ob eine Klasse threadsicher ist oder nicht, muss man sich über drei Eigenschaften von Operationen Gedanken machen:

atomar

- *Atomizität*: Welche Operationen sind überhaupt anfällig für Race Conditions? Hier ist wichtig, ob die Operation **atomar** (atomic) ist oder nicht: Ist sie atomar, wird sie entweder vollständig oder gar nicht ausgeführt. Eine Race Condition kann nicht auftreten. Ist die Operation nicht atomar, könne zwei Threads in Konflikt geraten.
- *Reihenfolge*: In welcher Reihenfolge werden Operationen ausgeführt? Die Reihenfolge im Quelltext entspricht auf modernen Computern nicht immer der Reihenfolge der Ausführung, daher braucht man eine Festlegung unter welchen Bedingungen die Reihenfolge beachtet wird.
- *Sichtbarkeit*: Wann sieht ein Thread Änderungen im Speicher? Durch Caches und Register gibt es mehrere Kopien der Daten im Speicher. Deswegen ist es wichtig zu definieren, wann eine Änderung in einer lokalen Kopie des Speichers durch einen Thread an andere Threads weitergegeben wird.

4.12 Atomare Operationen [67]

atomare Operation

- Eine **atomare Operation** kann nicht von einem anderen Thread gestört werden. Sie findet immer ganz oder gar nicht statt
- Ohne explizite Synchronisation sind nur folgende Operationen in Java atomar
  - Lesen eines volatilen 32bit Feldes (byte, short, int, float, char, boolean)

  - Schreiben eines volatilen 32bit Felds (byte, short, int, float, char, boolean)
- Nicht atomar sind
  - Lesen oder Schreiben eines 64bit Feldes (long, double)
  - Inkrementieren oder Dekrementieren eines 32bit Feldes
    `int i = 0; i++;`
- Es gibt spezielle Klassen, um komplexere atomare Operationen zu programmieren (`java.util.concurrent.atomic.*`)

Es gibt zwar einige Operationen in Java, die garantiert atomar sind, diese sind aber sehr begrenzt. So kann man zwar 32-Bit-Variablen atomar verändern aber diese Garantie reicht nicht aus, um damit threadsichere Programme zu schreiben. Bereits das Hochzählen der Variable ist nicht mehr atomar.

4.13 Ordering in Java [68]

within-thread as-if-serial

- Java fordert „**within-thread as-if-serial**“: Die Programmausführung und der Speicher müssen sich (in einem Threads) so verhalten, als ob die Instruktionen sequentiell ausgeführt würden
- Die VM darf beliebige Optimierungen machen, solange diese Bedingung erhalten bleibt

```
int i, k, m;                    int i, k, m;
i = 15;                         k = 18;
i++;                            i = 15;
k = 18;                         k++;
k++;                            i++;
m = i + k;                      m = i + k;
```

Die Java-VM darf die Reihenfolge, in der Instruktionen ausgeführt werden, beliebig ändern, solange für einen einzelnen Thread, der diese Instruktionen ausführt, keine Änderung sichtbar ist. D. h. für einen Thread fühlt es sich so an, als ob die Instruktionen seriell in der Reihenfolge des Programm-Sourcecodes ausgeführt würden (within-thread as-if-serial). Änderungen, die der Thread nicht bemerken kann, darf die VM aber beliebig ausführen.

Im vorliegenden Beispiel ist die Umsortierung für den ausführenden Thread nicht feststellbar, da auf die Variablen `k` und `i` erst bei der Zuweisung an `m` (`m = i + k`) wieder zugegriffen wird. Insofern darf die VM die Initialisierung und das Inkrementieren der Variablen in eine andere Reihenfolge bringen. Für den ausführenden Thread hat das keine Auswirkungen.

Greift aber ein anderer Thread parallel auf die Variablen zu, dann kann es passieren, dass die Umsortierung für diesen sichtbar wird, da die Garantie within-thread as-if-serial nur innerhalb eines Threads, nicht aber über Thread-Grenzen hinweg gilt. Ein Thread, der z. B. die Variablen `k` und `i` ausliest wird möglicherweise merken, dass `k` vor `i` erhöht wurde.

4.14 Was kommt hier raus? [69]

```
public class PossibleReordering {

    static int x = 0, y = 0, a = 0, b = 0;

    public static void main(String[] args) throws Exception {

        Thread one = new Thread(() -> { a = 1; x = b; });
        Thread two = new Thread(() -> { b = 1; y = a; });

        one.start(); two.start();
        one.join(); two.join();

        System.out.printf("x=%d, y=%d", x, y);
    }
}
```

```
Es kann alles rauskommen

x=0, y=0
x=1, y=0
x=0, y=1
x=1, y=1
```

Dieses Beispiel zeigt sehr gut die Effekte, die durch das Umsortieren der Instruktionen und das Scheduling auftreten können.

Dadurch, dass nicht festgelegt ist, welcher Thread wann gescheduled wird, kann sowohl Thread 1 (one) als auch Thread 2 (two) zuerst starten. Abhängig davon, welcher Thread zuerst zum Zuge kommt, wird das die Ausgabe x=0, y=1 (Thread 1 kam zuerst dran) oder x=1, y=0 (Thread 2 kam zuerst zum Zuge) sein.

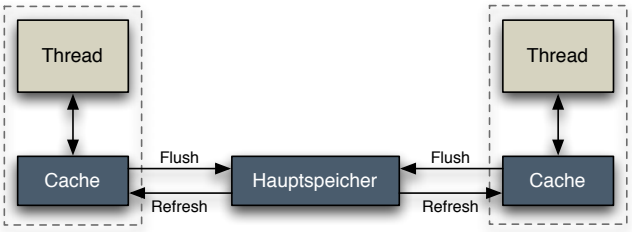
Durch das Umsortieren der Instruktionen innerhalb der Java-VM können aber noch andere Ergebnisse entstehen, nämlich x=0, y=0 und x=1, y=1. Da die Zuweisungen in den Threads (a=1, x=b) bzw. (b=1, y=a) jeweils unabhängig voneinander sind, kann die VM sie neu arrangieren zu (x=b, a=1) bzw. (y=a, b=1). Wenn jetzt ein Scheduling zwischen den Threads zwischen den Instruktionen erfolgt, können x=0, y=0 und x=1, y=1 als Ergebnis entstehen.

4.15 Das Java Memory Modell [71]

Memory Model

Das Java **Memory Model** definiert, wie sich die Java VM bezüglich des Speichers verhalten muss

- Plattformunabhängigkeit der Sprache
- Unterschiedliches Verhalten der Plattformen (z. B. IBM vs. Intel)



Als plattformunabhängige Sprache muss Java genau spezifizieren, wie sich der Speicher bei mehreren Threads verhalten soll. Diese Spezifikation wird als *Memory Model* bezeichnet. Da die Spezifikation sehr komplex und nicht eingängig ist, wollen wir im Folgenden mit einem vereinfachten mentalen Modell arbeiten. Das Modell entspricht zwar nicht zu 100% der Realität, ist aber geeignet alle Effekte und Regeln zu erklären.

Prinzipiell kann man sich vorstellen, dass jeder Thread einen eigenen Speicher hat, auf dem er ausschließlich arbeitet (den **Cache**). Er greift also nie direkt auf den Hauptspeicher zu, sondern benutzt nur seine lokale Kopie der Daten. Wenn jetzt zwei Threads gleichzeitig auf demselben Speicher arbeiten, stellt sich die Frage zu welchem Zeitpunkt Änderungen sichtbar werden. Um dies zu spezifizieren, definieren wir zwei Operationen:

Cache

- **Refresh**: Der Thread lädt den Inhalt seines Caches neu aus dem Hauptspeicher, d. h. die Daten im Cache werden durch die Daten aus dem Hauptspeicher ersetzt. Refresh
- **Flush**: Der Thread schreibt den Inhalt seines Caches zurück in den Hauptspeicher. Danach entspricht der Hauptspeicherinhalt dem Inhalt des Caches. Flush

Es ist offensichtlich, dass zwei Threads nur dann Daten miteinander austauschen können, wenn der eine Thread einen Flush macht und der andere *danach* einen Refresh. Aus diesem Grund ist es wichtig, zu definieren, wann diese Operationen erfolgen.

4.16 Garantien des Memory-Modells [72]

- Thread-Start / Thread-Ende
  - Start eines Threads → Refresh
  - Ende eines Threads → Flush
- Zugriff auf volatile Variablen
  - Lesen der Variable → Refresh
  - Schreiben der Variable → Flush
- Synchronisation auf einem Lock
  - Erhalt des Locks → Refresh
  - Freigeben des Locks → Flush

Das Java-Memory-Modell definiert, wann ein Flush und wann ein Refresh erfolgen muss. Offensichtlich ist, dass beim Start eines Threads ein Refersh und bei der Beendigung ein Flush erfolgt.

Die anderen Regeln werden wir im Folgenden noch erläutern. An dieser Stelle sei noch der Zugriff auf final Instanz-Variablen erläutert.

- Ende des Konstruktors → Partieller Flush
- Erstmaliges Lesen der Variable → Partieller Refresh

partiellen Flush

Bei dem **partiellen Flush** einer finalen Variablen wird das Objekt auf das die Variable verweist zuzüglich aller abhängigen Objekten (Objekten die von diesem aus erreichbar sind) in den Hauptspeicher geflusht.

partiellen Refresh

Bei einem **partiellen Refresh** einer finalen Variablen wird das Objekt auf das die Variable verweist zuzüglich aller abhängigen Objekten (Objekten die von diesem aus erreichbar sind) aus dem Hauptspeicher aufgefrischt.

4.17 Das Schlüsselwort volatile [73]

- Die Garantien des Java Memory Models gelten nur für einen Thread, nicht aber über Threads hinweg
- **volatile** erweitert die Garantien für eine Variable bei mehreren Threads, die auf diese Variable zugreifen

volatile

- Ein Thread, der von einer volatile Variablen liest, sieht immer den letzten und aktuellen Wert, den ein anderer Thread hineingeschrieben hat
- Für die Java VM bedeutet das
  - volatile Variablen dürfen *nicht in Registern abgelegt* werden
  - volatile Variablen dürfen *nicht anderweitig gecached* werden
  - volatile Variablen erzwingen eine **Memory Barrier**

Memory Barrier

Die Semantik von `volatile` erfordert sogar, dass alle anderen Variablen, die vor dem Schreiben der volatile Variablen geschrieben wurden ebenfalls für andere Threads sichtbar gemacht werden, sobald diese die volatile Variable gelesen haben, daher die Notwendigkeit einer Memory Barrier.

Man sollte trotzdem nicht zu viel auf `volatile` und dessen Semantik setzen, sondern eher andere Konstrukte, wie `synchronized` verwenden, da diese deutlich robuster sind.

Die Memory-Barrier hat dann zur Folge, dass beim Lesen der volatile-Variable ein Refresh und beim Schreiben ein Flush erfolgt.

4.18 Beispiel: volatile [74]

```
class Looper implements Runnable {
    static volatile boolean cont = true;
    public void run() {
```

```
        while (cont) {
            // mach was
        }
    }
}

public class LivelockFixed {

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new Looper());
        t.start();
        Thread.sleep(1000);
        Looper.cont = false;
    }
}
```

Durch das Hinzufügen von `volatile` zu `boolean cont = true;` verschwindet das Livelock aus dem vorhergehenden Beispiel. Jetzt sieht der Thread `Looper` die Änderung an der Variable durch den Main-Thread sofort.

4.19 Das Schlüsselwort synchronized [75]

- Wie sorge ich dafür, dass ein bestimmtes Code-Segment nur von einem einzigen Thread gleichzeitig durchlaufen werden kann?
- Benötigt wird ein Mechanismus, um Bereiche sperren (to lock) zu können, sobald ein Thread eingetreten ist und andere Threads damit auszusperren (**mutual exclusion**, **Mutex**)
- Das Schlüsselwort **synchronized** stellt diesen zur Verfügung
  - **synchronized Methoden**
  - **synchronized Blöcke**
- Jedes Java-Objekt kann für die **Sperre** (lock) zusammen mit `synchronized` verwendet werden

mutual exclusion
Mutex
synchronized
synchronized Methoden
synchronized Blöcke
Sperre

Die Java-Spezifikation spricht davon, dass es sich bei dem Java-Mechanismus für Sperren um so genannte **Monitore** (monitors) handelt: „a monitor is a concurrency construct that encapsulates data and functionality for allocating and releasing shared resources (such as network connections, memory buffers, printers, and so on)." Insofern kommt gelegentlich das Wort Monitor in der Literatur vor.

Monitore

Es ist an dieser Stelle wichtig zu unterscheiden zwischen:

- Dem *kritischen Abschnitt* also dem Bereich des Programms, der nur von einem Thread betreten werden soll und
- der *Sperre*, dem Objekt, das signalisiert, ob der Abschnitt belegt ist oder nicht.

Man kann sich die Beziehung zwischen kritischem Abschnitt und Sperre wie bei einer Straßenkreuzung vorstellen: Die Kreuzung ist der Kritische Abschnitt, zu keinem Zeitpunkt sollten Fahrzeuge aus zwei unterschiedlichen Richtungen in die Kreuzung einfahren. Das Lock ist aber nicht die Kreuzung selbst, sondern die Ampel, die signalisiert, ob die Kreuzung befahren werden darf oder nicht.

In Java ist jedes Java-Objekt in der Lage als Lock zu dienen. Hierzu hat es ein spezielles, aus Java nicht lesbares, „Attribut“, das anzeigt, ob der kritische Abschnitt frei ist oder nicht.

4.20 Beispiel: Ungeschützte Klasse [76]



```
public class SimpleStack {

    private Object[] stack;
    private int pos;

    public SimpleStack(int size) {
        stack = new Object[size];
        pos = 0;
    }

    public void push(Object o) {
        stack[pos++] = o;
    }

    public Object pop() {
       return stack[--pos];
    }
}
```

Die vorliegende Klasse ist nicht threadsicher, da sowohl die Operation `stack[pos++] = o` als auch `stack[--pos]` nicht atomar ist. Somit kann es passieren, dass Objekte auf dem Stack verloren gehen.

4.21 Beispiel: Synchronized bei Methoden [77]



```
public class SimpleStack {
    private Object[] stack;
    private int pos;

    public SimpleStack(int size) {
        stack = new Object[size];
        pos = 0;
    }

    public synchronized void push(Object o) {
```

```
        stack[pos++] = o;
    }

    public synchronized Object pop() {
       return stack[--pos];
    }
}
```

Durch die Markierung der Methoden mit `synchronized` werden diese als kritische Abschnitte definiert (*synchronized Methoden*). Die Folge ist, dass innerhalb der Methode immer nur ein Thread sein kann. Wichtig ist an dieser Stelle zu verstehen, dass es zwei kritische Abschnitte gibt (`push` und `pop`), beide aber über ein gemeinsames Lock geschützt werden (Details folgen). Dieses Lock ist `this`. Deswegen kann zu einem Zeitpunkt nur ein Thread in `push` oder `pop` sein. Dies ist an dieser Stelle richtig, da es andernfalls wieder zur parallelen Veränderung des Zählers `pos` kommen könnte.



```
public class SimpleStack {
    private Object[] stack;
    private int pos;

    public SimpleStack(int size) {
        stack = new Object[size];
        pos = 0;
    }
    public void push(Object o) {
        synchronized (this) {
            stack[pos++] = o;
        }
    }
    public Object pop() {
        synchronized (this) {
            return stack[--pos];
        }
    }
}
```

Anstatt die Methoden mit `synchronized` zu markieren, kann man *synchronized Blöcke* verwenden. Hier gibt man das Objekt, das als Lock dienen soll, explizit an. In diesem Fall wird `this` verwendet. Damit ist diese Klasse äquivalent zu der vorhergehenden, denn in Java ist



```
synchronized void methode() {
    // Kritischer Abschnitt
}
```

gleichbedeutend mit

```
void methode() {
    synchronized (this) {
        // Kritischer Abschnitt
    }
}
```

4.22 Beispiel: Synchronized Blöcke [79]

```
public class SimpleStack {
    private Object[] stack; private int pos;
    private final Object lock = new Object();
    public SimpleStack(int size) {
        stack = new Object[size];
        pos = 0;
    }
    public void push(Object o) {
        synchronized (lock) {
            stack[pos++] = o;
        }
    }
    public Object pop() {
       synchronized (lock) {
           return stack[--pos];
       }
    }
}
```

Man kann anstatt des eigenen Objektes jedes andere Objekt in einem synchronized-Block verwenden. Hierdurch hat man die Möglichkeit, innerhalb einer Klasse verschiedene kritische Abschnitte mit unterschiedlichen Sperren zu schützen. Dies erlaubt dann, dass Threads gleichzeitig in verschiedenen kritischen Abschnitten sind, solange diese unterschiedliche Locks verwenden.

4.23 Wie funktioniert synchronized? [80]

Grob kann man sich synchronized wie folgt vorstellen

Token

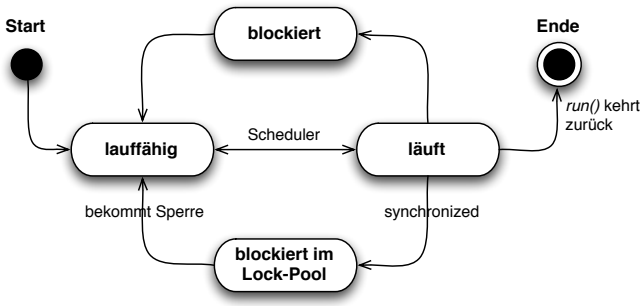
- Jedes Objekt in Java hat ein und nur ein spezielles **Token**
- Ein Thread, der in den synchronized Block (oder die Methode) eintreten will, braucht das Token vom Objekt
- Das Objekt „übergibt“ dem Thread das Token
- Ein Thread, der einen synchronized-Block (oder die Methode) verlässt gibt das Token zurück an das Objekt

Es handelt sich hierbei nur um eine Verdeutlichung, wie `synchronized` prinzipiell funktioniert. In der konkreten VM ist die Implementierung anders als hier dargestellt.

4.24 Wie funktionieren synchronized genau? [81]

Siehe externe Folien…

4.25 Synchronized im Zustandsdiagramm [82]



Die Verwendung von `synchronized` und kritischen Abschnitten führt zu einem weiteren Zustand im Thread-Zustandsdiagramm: **blockiert im Lock-Pool**. Wenn ein Thread in einen kritischen Abschnitt eintreten möchte, das Lock aber nicht bekommen kann, da der Abschnitt bereits belegt ist, wird der Thread in einen speziellen Pool gelegt, den **Lock-Pool**. Diesen Pool gibt es für jedes Lock, sodass beim Freiwerden des Locks einfach *zufällig* ein Thread aus dem Pool ausgewählt werden kann und das Lock bekommt und damit in den kritischen Abschnitt eintreten kann. Auch nach Erhalt des Locks muss der Thread aber noch warten, bis der Scheduler ihn laufen lässt – insofern geht er vom Lock-Pool in den Zustand lauffähig über.

blockiert im Lock-Pool

Lock-Pool

4.26 Java Locks sind reentrant [83]

Der synchronized-Mechanismus ist reentrant

- Hat ein Thread bereits das Token eines bestimmten Objektes, kann er in alle anderen synchronized-Blöcke eintreten, die von dem Objekt geschützt werden
- Ansonsten gäbe es permanent **Verklemmungen** (deadlocks)

Verklemmungen

```
public synchronized void push(Object o) {
    stack[pos++] = o;
}

public synchronized Object pop() {
   return stack[--pos];
}

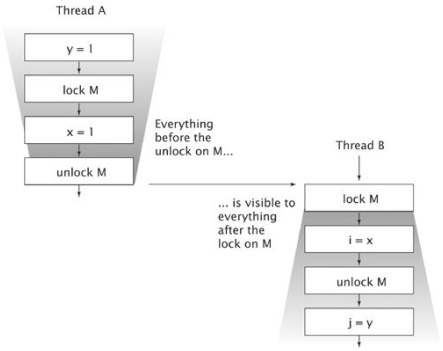
public synchronized void swap() {
```

```
    Object o1 = pop();
    Object o2 = pop();
    push(o1);
    push(o2);
}
```

4.27 Synchronized und das Memory Model [85]

Memory Barrier

- `synchronized` impliziert eine **Memory Barrier**
- Alle Variablen, die ein Thread geschrieben hat, bevor er den synchronized-Block eines Monitors M verlassen hat sind für alle anderen Threads sichtbar, die einen synchronized-Block für eben diesen Monitor betreten



Die Abbildung stammt aus der Java-Spezifikation und verwendet daher den Begriff *Monitor*.

Das Betreten eines synchronized-Blocks führt zu einem Refresh, das Verlassen zu einem Flush. Wenn also zwei Threads einen kritischen Abschnitt benutzten, der auf demselben Lock-Objekt synchronisiert ist (nicht zwingend denselben kritischen Abschnitt), sorgt diese Regel dafür, dass der zweite Thread den Speicherzustand sieht, den der erste Thread hinterlassen hat. Alle Änderungen des ersten Threads werden also für den zweiten sichtbar.

Wie man in der Grafik sieht, geht die Garantie sogar noch weiter: Variablen, die der erste Thread verändert hat *bevor* er den kritischen Abschnitt betreten hat, werden für den zweiten Thread sichtbar *nachdem* er einen kritischen Abschnitt mit demselben Lock betreten hat.

4.28 Synchronized und Leseoperationen [86]

Wegen der Auswirkungen auf die Sichtbarkeit von Variablen über Thread-Grenzen hinweg muss man `synchronized` nicht nur bei Schreib-, sondern auch bei Leseoperationen einsetzen

```
public synchronized void push(Object o) {
    stack[pos++] = o;
}

public synchronized Object pop() {
```

```
   return stack[--pos];
}

public Object peek() {
    return stack[pos - 1]; // FEHLER!!
}
```

`synchronized` hat also zwei Bedeutungen:

1. Es schützt kritische Abschnitte vor der parallelen Benutzung durch mehrere Threads.
2. Es sorgt über einen Flush-Refresh für die Sichtbarkeit von Änderungen im Speicher über Threadgrenzen hinweg.

Während die erste Regel eigentlich nur bei schreibenden Operationen relevant ist, führt die zweite dazu, dass man lesende Operationen mit `synchronized` schützen muss. Zwar benötigen diese gar nicht den Schutz von 1. würden dann aber wegen 2. auf möglicherweise veraltetem Speicher arbeiten. Denn die Garantie für die Sichtbarkeit von Speicheränderungen bezieht sich nur auf Threads die einen kritischen Abschnitt betreten, der durch dasselbe Lock geschützt ist. Gibt es keinen kritischen Abschnitt, gibt es keine Refresh und damit keine aktuellen Daten.

Aus diesem Grund ist das Beispiel falsch. Die lesende Methode `peek` muss ebenfalls durch ein `synchronized` geschützt werden, andernfalls sieht sie nicht den korrekten Zustand der Daten.

4.29 Lifelock und Synchonized [87]

```
class Looper implements Runnable {
    private boolean cont = true;
    public void requestStop() { cont = false; }
    public void run() {
        while (cont) { // FEHLER!!
            // mach was
        }
    }
}
public class Livelock {
    public static void main(String[] args) throws InterruptedException {
        Looper looper = new Looper();
        Thread t = new Thread(looper);
        t.start();
        Thread.sleep(1000);
        looper.requestStop();
    }
}
```

```
class Looper implements Runnable {
    private boolean cont = true;
    public synchronized void requestStop() { cont = false; }
```

```
    public void run() {
        while (cont) { // FEHLER!!
            // mach was
        }
    }
}
public class Livelock {
    public static void main(String[] args) throws InterruptedException {
        Looper looper = new Looper();
        Thread t = new Thread(looper);
        t.start();
        Thread.sleep(1000);
        looper.requestStop();
    }
}
```



```
class Looper implements Runnable {
    private boolean cont = true;
    public synchronized void requestStop() { cont = false; }
    private synchronized boolean cont() { return cont; }
    public void run() {
        while (cont()) {
            // mach was
        }
    }
}
public class FixedLiveLock {
    public static void main(String[] args) throws InterruptedException {
        Looper looper = new Looper();
        Thread t = new Thread(looper);
        t.start(); Thread.sleep(1000);
        looper.requestStop();
    }
}
```

4.30 Deadlocks [90]



- Was passiert, wenn zwei Threads gegenseitig auf die Freigabe eines Locks durch den jeweils anderen Thread warten?
- Es kommt zur **Verklemmung** (deadlock) (Liveness Hazard) Verklemmung
  - Die Threads können nicht weiterlaufen, sie blockieren sich gegenseitig
  - Die VM wird die Verklemmung nicht automatisch auflösen
  - Manchmal kann man einen blockierten Thread durch `interrupt()` wecken
- Gegenmaßnahmen
  - Locks immer in derselben Reihenfolge nutzen

Der Zustand eines Deadlocks kann folgendermaßen definiert werden: Eine Menge von Threads befindet sich in einem Deadlock, wenn jeder dieser Threads auf ein Ereignis wartet, das nur ein anderer Threads aus dieser Menge verursachen kann.

4.31 Beispiel: Deadlock [92]

```
class T1 implements Runnable {
    public void run() {
        synchronized (Deadlock.lock1) {
            Thread.yield();
            synchronized (Deadlock.lock2) {
            }
```

```
        }
    }
}
class T2 implements Runnable {
    public void run() {
        synchronized (Deadlock.lock2) {
            Thread.yield();
            synchronized (Deadlock.lock1) {
            }
        }
    }
}
```

```
class ClassForLock1 { /* empty */ }
class ClassForLock2 { /* empty */ }

public class Deadlock {

    public static Object lock1 = new ClassForLock1();
    public static Object lock2 = new ClassForLock2();

    public static void main(String[] args) {
        Thread t1 = new Thread(new T1());
        Thread t2 = new Thread(new T2());

        t1.start();
        t2.start();
    }
}
```

4.32 Tipp: Dedizierte Lock-Klassen [94]

- Wenn ein Deadlock auftritt ist es wichtig zu wissen, welcher Thread auf welches Lock wartet
- Verwendet man `Object` als Klasse für die Lock-Objekte bekommt man nur dürftige Informationen
- Durch dedizierte Klassen für die Locks kann man die Fehlersuche deutlich verbessern





Die Java-VM ist in der Lage bei einem Deadlock anzuzeigen, welcher Thread auf welches Lock wartet. Hierzu dient das Werkzeug `jconsole` oder der Debugger von Eclipse. Hierbei ist es nützlich, wenn man den Locks sinnvolle Namen gibt, da man andernfalls nur Ausgaben der Form `Object@138423` bekommt. Zu diesem Zweck benutzt man einfach statische innere Klassen, deren Namen einen Hinweis auf den Zweck des Locks gibt.

4.33 Thread Local Storage [96]

- Der Heap wird von allen Threads gemeinsam genutzt
- Der Stack ist getrennt für die Threads
- Wie kann man Heap-Speicher für einen einzelnen Thread reservieren?
- Thread Local Storage erlaubt es, Speicher pro Thread auf dem Heap zu halten. Hierzu dient die Klasse ThreadLocal
  - `public Object get()` – Liest die Variable aus
  - `public void set(Object value)` – Setzt die Variable
  - `public void remove()` – Entfernt die Daten aus dem ThreadLocal

Tatsächlich ist `ThreadLocal` inzwischen ein generischer Typ, sodass man nicht mehr mit `Object` und Casts hantieren muss. Korrekt lautet sie inzwischen:

- `public T get()`
- `public void set(T value)`
- `public void remove()`

4.34 Beispiel: ThreadLocal [97]

```
public class Daten {
    public static final ThreadLocal<Integer> tl = new ThreadLocal<Integer>();
}

public class ThreadLocalDemo implements Runnable {

    public void run() {
        Daten.tl.set(0);

        while (true) {
            int wert = Daten.tl.get();
```

```
            Daten.tl.set(++wert);
            System.out.printf("Thread %s: Wert=%d%n",
                    Thread.currentThread().getName(), wert);
            Thread.yield();
        }
    }
}
```

```
Runnable runnable = new ThreadLocalDemo();
new Thread(runnable, "1").start();
new Thread(runnable, "2").start();
```

```
...
Thread 1: Wert=10280
Thread 2: Wert=9876
Thread 1: Wert=10281
Thread 2: Wert=9877
Thread 1: Wert=10282
Thread 2: Wert=9878
Thread 1: Wert=10283
Thread 2: Wert=9879
...
```

4.35 Amdahl's Law [99]

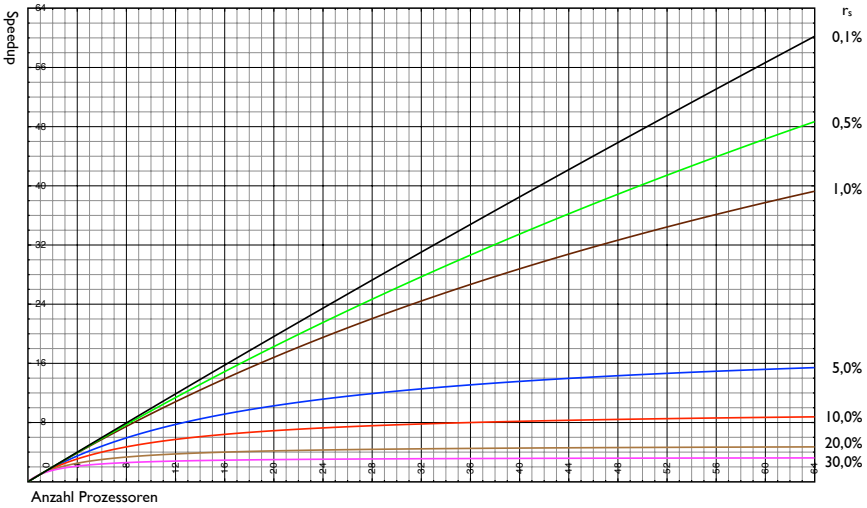
Der maximale Geschwindigkeitsgewinn (Speedup) durch Parallelprogrammierung wird durch Amdahl's Law nach oben begrenzt

- r_s – Nicht parallelisierbarer Anteil des Programms (sequentiell)
- r_p – Parallelisierbarer Anteil des Programms (parallel)
- n – Anzahl der Prozessoren

$$Speedup = \frac{1}{(r_s + \frac{r_p}{n})}, r_s + r_p = 1$$

Quelle: Gene M. Amdahl, Validity of the single processor approach to achieve large scale computing capabilities, AFPIS spring joint computer conference, 1967

http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf


Speedup
64
56
48
40
32
24
16
8
0
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
Anzahl Prozessoren
r_s
0,1%
0,5%
1,0%
5,0%
10,0%
20,0%
30,0%

Kapitel 5

Thread-Koordination

5.1 Das Problem des schlafenden Friseurs [102]



Quelle: Dr. Henrik Brosenne, Universität Göttingen

Das Problem des schlafenden Friseurs

- In dem Friseurladen arbeitet 1 Friseur
- Es gibt 1 Friseurstuhl und n Stühle für wartende Kunden
- Falls kein Kunde anwesend ist, setzt sich der Friseur auf seinen Friseurstuhl und schläft.
- Ein neuer Kunde muss den Friseur aufwecken
- Weitere Kunden nehmen auf einem der n Stühle Platz oder verlassen den Laden falls alle Stühle besetzt sind.

5.2 Das Problem [103]

Producer
Consumer

- Ein Thread (**Producer**) erzeugt Daten und stellt sie in eine Queue ein anderer Thread (**Consumer**) soll sie verarbeiten und aus der Queue herausnehmen
- Wie vermeidet man, dass der Consumer ständig nachfragen muss, ob etwas für ihn vorliegt?
- Wie signalisiert man dem Consumer-Thread, dass Arbeit in der Queue vorhanden ist?
- Wie sorgt man dafür, dass der Producer-Thread aufhört Daten in die Queue zu stellen, wenn der Consumer-Thread nicht mehr nachkommt?

Man benötigt eine Lösung, die folgendes bietet:

- Ein Thread kann sich schlafen legen
- Ein anderer Thread kann diesen Thread wieder aufwecken

Der *Consumer* legt sich schlafen wenn die Queue leer ist und weckt andere Threads auf, wenn er etwas aus der Queue genommen hat.

Der *Producer* legt sich schlafen wenn die Queue voll ist und weckt andere Threads auf, wenn er etwas in die Queue hineingelegt hat.

5.3 Die Lösung: wait() und notify() [104]

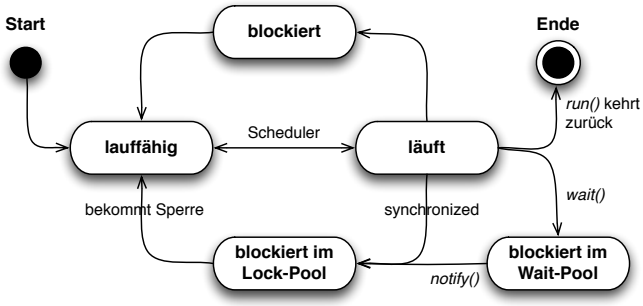
- Die Klasse `Object` bietet zwei Methoden an, um das beschriebene Problem zu lösen
  - `wait()` – legt einen Thread schlafen
  - `notify()` – weckt einen der Threads, die sich mit `wait()` schlafen gelegt haben wieder auf
  - `notifyAll()` – weckt alle wartenden Threads wieder auf
- `wait()`, `notify()` und `notifyAll()` dürfen nur auf Objekten aufgerufen werden für die der Thread das Lock hält
- `wait()` sollte immer in einer Schleife verwendet werden

`wait()`

`notify()`

`notifyAll()`

5.4 Zustandsdiagramm mit Wait-Pool [105]



Durch `wait()` und `notify()` kommt ein neuer (und der letzte) Zustand bei Threads hinzu: **blockiert im Wait-Pool**. Wenn ein Thread auf einem Objekt `wait()` aufruft, so wandert er in diesen Zustand. Das Lock gibt er zurück, obwohl der mitten in einem `synchronized` steht. Dies ist eine Besonderheit von `wait()`: Ein Aufruf der Methode gibt das Token zurück.

blockiert im Wait-Pool

Der Thread verbleibt jetzt solange im *Wait-Pool*, bis ein anderer Thread auf demselben Objekt `notify()` oder `notifyAll()` aufruft. Bei `notify()` wandert ein (zufällig ausgesuchter) Thread in den Lock-Pool, bei `notifyAll()` alle. Da der Thread, der `notify()` aufgerufen hat, in diesem Moment noch das Lock hält, bleiben die geweckten Threads solange im Lock-Pool, bis einer von ihnen das Lock bekommen kann, weil der andere Thread es beim Verlassen des kritischen Abschnitts zurückgegeben hat.

5.5 Beispiel: Producer / Consumer [106]

```
public class Producer implements Runnable {
    Queue<String> queue;

    public Producer(Queue<String> queue) {
        this.queue = queue;
    }
    public void run() { ... }
}

public class Consumer implements Runnable {
    Queue<String> queue;

    public Consumer(Queue<String> queue) {
        this.queue = queue;
    }

    public void run() { ... }
}
```

5.6 Beispiel: Producer – run()-Methode [107]

```
int i = 0;
try {
    while (true) {
        synchronized (queue) {

            while (queue.size() > 10) {
                queue.wait();
            }

            queue.add("P" + ++i);
            queue.notifyAll();
        }

        Thread.sleep(100);
    }
}
catch (InterruptedException e) {
}
```

Normalerweise würde man den Code, der die Größe der Queue testet und entsprechend das `wait()` bzw. `notify()` macht in die Queue selbst verlagern, da die hier durchgeführte Synchronisation von außen sehr fehleranfällig ist.

5.7 Beispiel: Consumer – run()-Methode [108]

```
try {
    while (true) {
        synchronized (queue) {

            while (queue.isEmpty()) {
                queue.wait();
            }
            String element = queue.poll();
            queue.notifyAll();

            System.out.println(Thread.currentThread().getName()
                    + ": " + element);
        }
        Thread.sleep(100);
    }
}
catch (InterruptedException e) {
}
```

5.8 Beispiel: Start-Code [109]

```
Queue<String> queue = new LinkedList<String>();

Producer producer = new Producer(queue);
Consumer consumer = new Consumer(queue);

Thread tp = new Thread(producer, "Producer");
Thread c1 = new Thread(consumer, "Consumer 1");
Thread c2 = new Thread(consumer, "Consumer 2");
Thread c3 = new Thread(consumer, "Consumer 3");

tp.start();
c1.start();
c3.start();
c2.start();
```

```
Consumer 1: P1
Consumer 2: P2
Consumer 1: P3
Consumer 2: P4
Consumer 1: P5
Consumer 3: P6
```

5.9 Wie funktionieren wait() und notify()? [111]

Siehe externe Folien…

5.10 Was muss ich mir merken… [112]

- Man kann `wait()` und `notify()` auf jedem beliebigen Objekten verwenden
- Man darf die Methoden nur in einem synchronized-Block verwenden, der *dasselbe* Objekt als Lock verwendet
- Test, der zum `wait()` führt sollte immer in einer Schleife ausgeführt werden, da er noch einmal wiederholt werden muss – der Thread weiß nicht warum er geweckt wurde
- Bei Producer-Consumer muss man immer `notifyAll()` verwenden, da `notify()` zum Verhungern führen kann

Kapitel 6

java.util.concurrent

6.1 Timer und TimerTask [114]

- **Timer** erlaubt die Ausführung von nebenläufigen Aktionen — Timer
- **TimerTask** stellt den Code zur Verfügung, der ausgeführt werden soll — TimerTask
- Anwender muss sich nicht um Threads kümmern, das übernimmt die Klasse `Timer`, die einen Thread für die Ausführung der Tasks verwendet
- `Timer` kommt mit sehr vielen (> 1000) Tasks gut zu recht
- Tasks dürfen nicht zu lange laufen, da es sonst zu Staus kommt

6.2 Beispiel: Timer / TimerTask [115]

```
public class Timebomb extends TimerTask {
    public void run() {
        System.out.print("Boom!");
        cancel();
    }
}
```

```
Timer timer = new Timer("Timebomb", true);
timer.schedule(new Timebomb(), 3000);
for (int i = 0; i < 1000; i++) {
    System.out.print(i + " ");
    Thread.sleep(1000);
}
timer.cancel();
```

```
0 1 2 3 Boom! 4 5
```

```
public class Countdown extends TimerTask {
    private int zaehler;
    public Countdown(int zaehler) { this.zaehler = zaehler; }

    public void run() {
        if (zaehler-- <= 0) {
            System.out.print("Boom!");
            cancel();
        }
    }
}
```

```
Timer timer = new Timer("Countdown");
timer.scheduleAtFixedRate(new Countdown(4), 0, 500);

for (int i = 0; i < 10; i++) {
    System.out.print(i + " ");
    Thread.sleep(1000);
}

timer.cancel();
```

```
0 1 Boom! 2 3 4
```

6.3 java.util.concurrent [118]

Seit Java 5 gibt es fertige Lösungen für viele Thread-Probleme in `java.util.concurrent`

- Producer/Consumer mit `BlockingQueue`
- Performante, threadsichere Collections
  - `ConcurrentHashMap`
  - `CopyOnWriteArrayList`
- Feingranulare Thread-Synchronisation und Steuerung
  - `Barrier`, `Latch`, `Semaphore`
- Ausführung paralleler Tasks
  - `Executor`
  - `Future`

6.4 Producer/Consumer mit BlockingQueue [119]

`BlockingQueue<E>` enthält bereits alle Zutaten um ein Producer/Consumer-Pattern zu implementieren

- `boolean offer(E e)` – Versucht ein Element hinzuzufügen, ist die Queue voll, wird false zurückgegeben
- `boolean offer(E e, long timeout, TimeUnit unit)` – Wie `offer`, nur mit timeout
- `void put(E e)` – Versucht ein Element hinzuzufügen. Ist die Queue voll, blockiert die Methode solange bis wieder Platz in der Queue ist
- `E take()` – Versucht ein Element aus der Queue zu nehmen, ist sie leer, blockiert die Methode
- `E poll(long timeout, TimeUnit unit)` – Wie `take`, nur mit Timeout
- `int remainingCapacity()` – Verbleibende Kapazität auf der Queue

6.5 Beispiel: BlockingQueue (Producer) [120]

```
public class Producer implements Runnable {
    BlockingQueue<String> queue;
    ...

    public void run() {
        int i = 0;

        try {
            while (true) {
                queue.put("P" + ++i);
                Thread.sleep(100);
            }
        }
        catch (InterruptedException e) {
        }
    }
}
```

6.6 Beispiel: BlockingQueue (Consumer) [121]

```
public class Consumer implements Runnable {
    BlockingQueue<String> queue;
    ...

    public void run() {
        try {
            while (true) {
                String element = queue.take();
```

```
                System.out.println(Thread.currentThread().getName()
                        + ": " + element);
                Thread.sleep(100);
            }
        }
        catch (InterruptedException e) {
        }
    }
}
```

6.7 CountDownLatch [122]

- Ein Latch erlaubt es Threads darauf warten zu lassen, dass ein bestimmter Zustand des Latches eintritt
- Ein CountDownLatch wird mit einem bestimmten Zahlenwert initialisiert und Threads können ihn herunter zählen. Wenn 0 erreicht ist, laufen all Threads weiter
  - `void await()` – Thread blockiert bis der Zähler den Wert 0 erreicht hat
  - `boolean await(long timeout, TimeUnit unit)` – Thread blockiert bis der Zähler den Wert 0 erreicht hat oder timeout verstrichen ist
  - `void countDown()` – Der Zähler wird um eins herunter gezählt
- Der Zähler kann nicht zurückgesetzt werden

6.8 Beispiel: Threads vereinigen mit Latch [123]

```
public class Join {

    static CountDownLatch startLatch = new CountDownLatch(1);
    static CountDownLatch endLatch = new CountDownLatch(3);

    static class Runner implements Runnable {
        String text;

        Runner(String text) {
            this.text = text;
        }

        public void run() {
            ...
        }
    }
}
```

```
public void run() {
   try {
```

```
        startLatch.await();

        for (int i = 0; i < 5; i++) {
            System.out.println(text);
            Thread.yield();
        }
        System.out.println(text + " ist fertig.");
        endLatch.countDown();
    } catch (InterruptedException e) { // ignorieren
    }
}
```

```
new Thread(new Runner("Runner 1")).start();
new Thread(new Runner("Runner 2")).start();
new Thread(new Runner("Runner 3")).start();
startLatch.countDown(); // alle laufen los
endLatch.await();     // warten bis alle fertig sind
System.out.println("Alle fertig");
```



```
Runner 3
Runner 2
Runner 1
Runner 3
Runner 2
Runner 1 ist fertig.
Runner 3
Runner 3 ist fertig.
Runner 2
Runner 2 ist fertig.
Alle fertig
```


6.9 Executor [126]

- Führt Tasks in eigenen Threads aus
- Übernimmt das Pooling der Threads
- Erlaubt Tasks für spätere Zeiträume einzuplanen

6.10 Beispiel: Executor [127]

```
class Task implements Runnable {
    String name;
    public Task(String name) { this.name = name; }

    public void run() {
```

```
        for (int i = 0; i < 100; i++) {
            System.out.println(name);
            Thread.yield();
        }
    }
}
```

```
ExecutorService ex = Executors.newCachedThreadPool();
ex.execute(new Task("Task 1"));
ex.execute(new Task("Task 2"));

ScheduledExecutorService ex2 = Executors.newScheduledThreadPool(3);

ex2.schedule(new Task("Delayed 1"), 5, TimeUnit.SECONDS);
ex2.schedule(new Task("Delayed 2"), 10, TimeUnit.SECONDS);
```

Weggelassen sind im Beispiel die Imports:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
```

6.11 Future [129]

Berechnung deren Ergebnis zu einem späteren Zeitpunkt benötigt wird

- `boolean cancel(boolean mayInterruptIfRunning)` – Versucht die Ausführung abzubrechen bzw. zu verhindern
- `boolean isCancelled()` – Zeigt an, ob der Task abgebrochen wurde bevor er zu Ende laufen konnte
- `boolean isDone()` – Zeigt an, ob der Task beendet wurde (auch unterbrochen etc.)
- `V get()` – Holt das Ergebnis ab und blockiert solange, bis das Ergebnis vorhanden ist, d. h. der Task beendet wurde
- `V get(long timeout, TimeUnit unit)` – Wie `get()` wartet aber maximal für das angegebene Timeout

6.12 Beispiel: Callable [130]

```
class RechnerCallable implements Callable<Integer> {
    public Integer call() throws Exception {
        return 2 + 2;
```

```
    }
}
```

```
class RechnerRunnable implements Runnable {
    private int result;
    private Exception e;

    public int getResult() { return result; }
    public Exception getException() { return e; }

    public void run() {
        try {
            result = 2+2;
        }
        catch (Exception e) {
            this.e = e;
        }
    }
}
```

6.13 Beispiel: Future [132]

```
class FutureCallable implements Callable<String> {
    public String call() throws Exception {
        Thread.sleep(1000); // unglaublich teure Berechnung ;-)
        return "Ergebnis";
    }
}
```

```
ExecutorService executor = Executors.newCachedThreadPool();
Future<String> future = executor.submit(new FutureCallable());
System.out.println("Viele interessante Dinge parallel");

try {
    String ergebnis = future.get();
    System.out.println(ergebnis);
}
catch (ExecutionException ex) { /* oops */ }
System.out.println("Alles fertig");
```

6.14 Atomics [133]

Komplexere atomare Operationen können mit java.util.concurrent.atomic gebaut werden

- Primitive Typen: `AtomicBoolean`, `AtomicInteger`, `AtomicLong`
- Arrays: `AtomicIntegerArray`, `AtomicLongArray`
- Referenzen: `AtomicReference`, `AtomicReferenceArray`

```
public class AtomicExample implements Runnable {
    private final AtomicInteger counter = new AtomicInteger(1);
    public void run() {
        while (counter.getAndIncrement() > 0) {
            System.out.printf("Wert: %d%n", counter.get());
            Thread.yield();
        }
        System.out.println("** Gestoppt **");
    }

    public static void main(String[] args) throws Exception {
        AtomicExample st = new AtomicExample();
        Thread thread = new Thread(st);
        thread.start();
        Thread.sleep(100);
        st.counter.set(-1);
    }
}
```

Index

ALU, 24
Arithmetisch-logische Einheit, 24
atomar, 29
atomare Operation, 29
Atomizität, 29

blockiert, 15
blockiert im Lock-Pool, 38
blockiert im Wait-Pool, 48

Cache, 32
check-then-act, 28
Consumer, 47
Control Unit, 24

deadlock, 26

Flush, 32

green threading, 8

Input Output Unit, 24
Interprozess-Kommunikation, 6
interruptible, 19
IPC, 6

Kerne, 5
kooperativ, 5
kooperatives Scheduling, 8

lauffähig, 15
livelock, 26
Liveness Hazards, 26
Lock-Pool, 38
läuft, 15

Memory Barrier, 33, 39
Memory Model, 31
Monitore, 34
Moores Gesetz, 3
Multi-Core-Prozessoren, 5
Mutex, 34
mutual exclusion, 34

native threads, 7
nicht unterbrechende, 5
NUMA, 25

partiellen Flush, 33
partiellen Refresh, 33
Performance Hazards, 26
preemptives Scheduling, 7
Producer, 47
Prozesse, 5

race condition, 26
Refresh, 32
Register, 24
Reihenfolge, 29

Safety Hazards, 26
Scheduler, 5
Sichtbarkeit, 29
Sperre, 34
starvation, 26
synchronized, 34
synchronized Blöcke, 34
synchronized Methoden, 34

Threads, 5
threadsicher, 29
Timer, 52

TimerTask, 52
Token, 37

unterbrechende, 5

Verklemmung, 42
Verklemmungen, 38
volatile, 33

within-thread as-if-serial, 30

Zeitscheiben, 7