



# Java Einführung

Ausgabe 08.2020

Copyright © 2011 – 2020, thomas.iten@iten-engineering.ch

Alle Rechte vorbehalten.  
Reproduktion (auch auszugsweise) ist nur mit schriftlicher Bewilligung des Verfassers gestattet.

1

## Inhalt

Kapitel	Inhalt	Seite
<b>Teil I</b>	<b>Basiswissen</b>	
1	About	4
2	Write Once, Run Anywhere	6
3	Hello World	41
4	Grundlegende Sprachelemente	48
5	Kontrollstrukturen	97
<b>Teil II</b>	<b>Objektorientierung</b>	
6	Klassen, Attribute und Methoden	107
7	Kapselung und Konstruktoren	135
8	Vererbung	151
9	Packages	192
10	Interfaces und Adapterklassen	205

2

## Inhalt II

Kapitel	Inhalt	Seite
<b>Teil III</b>	<b>Weitere Datentypen</b>	
11	Strings und Wrapper Klassen	232
12	Arrays, Varargs und Enum	249
13	Collection Framework	287
<b>Teil IV</b>	<b>Fehlerbehandlung</b>	
14	Exceptions	338
15	Assertions und Annotations	361
<b>Teil V</b>	<b>Weitere APIs</b>	
16	Dateien	374
17	Streams	407
18	Random, Date & Time, System, Console	450
19	Funktionale Programmierung	462
<b>Anhang</b>	<b>Literatur und WebLinks</b>	482

## Kapitel 1

### About

## About

- ▶ Der Kurs vermittelt Ihnen eine umfangreiche Einführung in die Java Programmiersprache und eine Übersicht über die vielen Einsatzmöglichkeiten.
- ▶ Sie verstehen Javas Systemarchitektur und kennen die fundamentalen Klassen und Sprachelemente.
- ▶ Nach dem Kurs sind Sie in der Lage, selbständig einfache Java-Programme zu schreiben.

## Kapitel 2

**Write Once, Run Anywhere**

## Abschnitt I

### Übersicht

# Write Once, Run Anywhere !



Wie funktioniert das?

## Java Technologien

- Die Java Technologien bilden eine standardisierte Software Plattform mit: **Programmiersprache**, **Entwicklungswerkzeugen** und **Laufzeitumgebung**



Quelle: <http://de.wikipedia.org>

## Programmiersprache

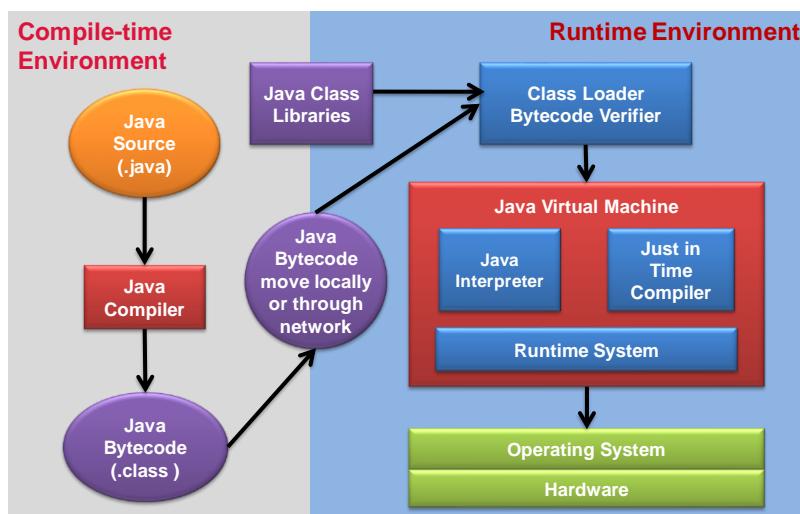
- Java ist eine moderne, **plattformunabhängige**, **objektorientierte** Programmiersprache
- Java bietet u. a. folgende Features:
  - automatic type checking  
(Java ist streng typisiert)
  - automatic garbage collection  
(es gibt keine Destruktoren)
  - simplified pointers  
(direkte Zugriffe aufs Memory sind nicht möglich)
  - simplified network access
  - multi-threading

## Entwurfsziele

Der Entwurf der Programmiersprache strebte im u.a. folgende Ziele an:

- ▶ einfache, objektorientierte und „vertraute“ Programmiersprache für verteilte Systeme
- ▶ robust und sicher
- ▶ architekturneutral und portabel
- ▶ sehr leistungsfähig
- ▶ interpretierbar, parallelisierbar und dynamisch

## und so funktioniert's



## Java Editionen

- ▶ Java Micro Edition & Java Card Technologie
  - Technologien für «mobile» und «embedded» Geräte sowie Geräte mit limitierten Speicher und Prozessor Ressourcen wie zum Beispiel «smart cards»
- ▶ Java Standard Edition
  - Technologie für Desktop und Server Anwendungen
- ▶ Java Enterprise Edition
  - Industrie Standard für mehrschichtige (Multi Tier) Anwendungen
  - Erstellung von portablen, robusten, skalierbaren und sicheren Server Applikationen

## Java Micro Edition (Java ME) und Java Card Technologie

- ▶ Robuste und flexible Technologie für „embedded“ und „mobile“ Geräte wie zum Beispiel: Micro Controller, Sensoren, mobile Phone & PDAs, TV SetTop Boxen, Drucker, etc.
- ▶ Unterteilung in Connected Devices (CDC), Connected Limited Devices (CLDC) sowie Java Card Technologien

Java for  
Mobile DevicesJava  
EmbeddedJava  
TVJava Card  
Technology

## Abschnitt II

### Java Standard Edition

### Java Standard Edition (Java SE)

- ▶ Besteht grundsätzlich aus den beiden Produkten:
  - Java Runtime Environment (JRE)
  - Java Development Kit (JDK)
- ▶ Die JRE beinhaltet alle notwendigen Komponenten für die Laufzeitumgebung
- ▶ Das JDK ist ein Super Set der JRE mit zusätzlichen Tools, Compiler und Debugger
- ▶ Unterstützt Web Services
- ▶ Bildet Grundlage für die Java Enterprise Edition

## Java SE Platform at a Glance

Quelle: <https://www.oracle.com/java/technologies/platform-glance.html>

Copyright © iten-engineering.ch

Java Einführung

17

17

Jahr	Version	Neue Features
1996	1.0	Applets, AWT, I/O, Net
1997	1.1	RMI, JDBC, etc.
1998	1.2	Swing, Collections, etc.
2000	1.3	CORBA ORB, Sound, Servlets, etc.
2002	1.4	XML Processing, JNI, etc.
2004	1.5	Generics, Enhanced for Loop, Autoboxing, Enums, Annotations, etc.
2006	6	Scripting, Compiler Access, Desktop Deployment...
2011	7	Byte Code Enhancement, NIO, Strings in Switch, try-close, Unicode 6, etc.
2014	8	Lambda Expressions, JavaScript Runtime, new Date and Time API, Java FX
2017	9	Modularization, Reactive Stream Flows, Collection Factory Methods
2018	10	Release 18.3 NON LTS (Long Time Support)
2018	11	<b>Release 18.9 LTS (Long Time Support)</b>
2019	12, 13	Switch Expression & Text Blocks (Preview), Low Pause Time Garbage Collection, etc.
2020	14	Switch Expression (Standard), Records (Preview), etc.

Geschichte

Oracle hat einen neuen Release Zyklus angekündigt:

- Feature Release alle 6 Monate, Maintenance Release alle 3 Monate
- Die Version wird ergänzt mit Angabe: JJ.MM, Support Release bis zum nächsten Feature Release
- **Alle 3 Jahre erscheint eine LTS (Long Term Support) Version**

Quelle: [https://en.wikipedia.org/wiki/Java\\_version\\_history](https://en.wikipedia.org/wiki/Java_version_history)

Copyright © iten-engineering.ch

Java Einführung

18

18

## Abschnitt III

### Java Enterprise Edition

### Java Enterprise Edition (Java EE)

- ▶ Mit dem “Java EE Application Model” werden die Aufgaben zur Erstellung einer mehrschichtigen Anwendung unterteilt:
  - Die Geschäfts- und Präsentationslogik wird durch den Entwickler erstellt
  - Die Standard System Services stellt die Java EE Platform zur Verfügung
- ▶ Der Entwickler kann sich darauf verlassen, dass die komplexen “System Level” Aufgaben bei mehrschichtigen Anwendungen durch die Plattform gelöst werden.

## Multi Tier Architektur

The diagram illustrates a Multi Tier Architecture with the following layers from top to bottom:

- Client Tier:** Contains "Java EE Application 1" (Application Client) and "Java EE Application 2" (Web Pages).
- Web Tier:** Contains "JavaServer Faces Pages".
- Business Tier:** Contains "Enterprise Beans" (represented by coffee beans).
- EIS Tier:** Contains "Database" components.

Arrows indicate the flow from the Client Tier down through the Web Tier, Business Tier, and finally to the Database Server in the EIS Tier. A "Java EE Server" box encloses the Web Tier and Business Tier.

Java EE definiert die folgenden Komponenten:

- ▶ Application Clients und Applets
- ▶ Java Servlet, JavaServer Faces und JavaServer Pages
- ▶ Enterprise JavaBeans

Quelle: Oracle, Java EE 7 Tutorial  
Copyright © iten-engineering.ch      Java Einführung      21

21

## Multi Tier Architektur II

- ▶ Unterteilung der Applikationslogik in Komponenten entsprechend Ihrer Funktion
  - Präsentation
  - Geschäftslogik
  - Datenhaltung
- ▶ Verteilung / Installation der Komponenten auf verschiedenen Rechnern
- ▶ 3 vs. 4 Tier gemäss Oracle (Sun Microsystems):
 

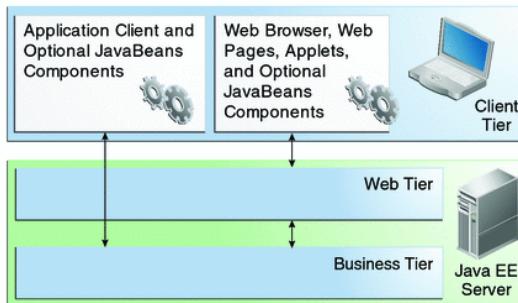
Although a Java EE application can consist of the three or four tiers, Java EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations.

Copyright © iten-engineering.ch      Java Einführung      22

22

## Client Tier

- ▶ Standalone Clients (Java oder andere Technologien) die auf den Web oder Business Tier zugreifen
- ▶ Web Clients / Applets die auf den Web Tier zugreifen



Quelle: Oracle, Java EE 7 Tutorial

Copyright © iten-engineering.ch

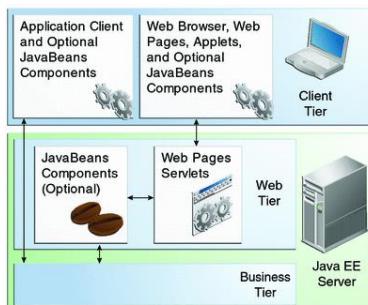
Java Einführung

23

23

## Web Tier

- ▶ Interaktive Web Applikationen mit dynamischen Inhalten
- ▶ Service orientierte Applikationen (Web Services) welche Business Dienste zur Verfügung stellen



Java Servlets, Java Server Faces und Java Server Pages Komponenten

Quelle: Oracle, Java EE 7 Tutorial

Copyright © iten-engineering.ch

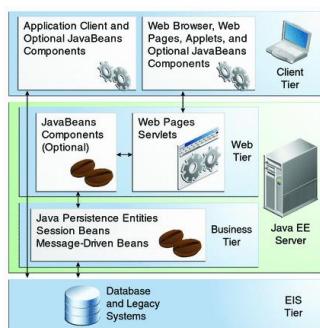
Java Einführung

24

24

## Business Tier

- ▶ Enterprise JavaBean Komponenten
  - Verwaltung durch Java EE Server
  - Der Server überprüft ob die Komponenten konform mit der Spezifikation sind



Session Beans, Message  
Driven Beans und Java  
Persistence Entities

Quelle: Oracle, Java EE 7 Tutorial

Copyright © iten-engineering.ch

Java Einführung

25

25

## Container

- ▶ Die verschiedene Komponenten werden in sogenannten Containern ausgeführt
- ▶ Diese bilden die Schnittstelle zwischen der Komponente und den Plattform spezifischen „low level“ Funktionen
- ▶ Bevor eine Komponente ausgeführt wird muss sie entsprechend zusammengestellt (assembliert) und in den Container ausgeliefert (deployed) werden

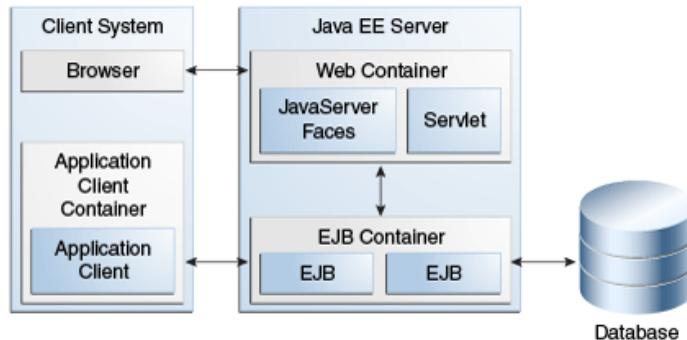
Copyright © iten-engineering.ch

Java Einführung

26

26

## Container Typen



- ▶ **Java SE Container:**
  - Application Client Container
  - Applet Container
- ▶ **Java EE Container:**
  - Web Container
  - EJB Container

Quelle: Oracle, Java EE 7 Tutorial

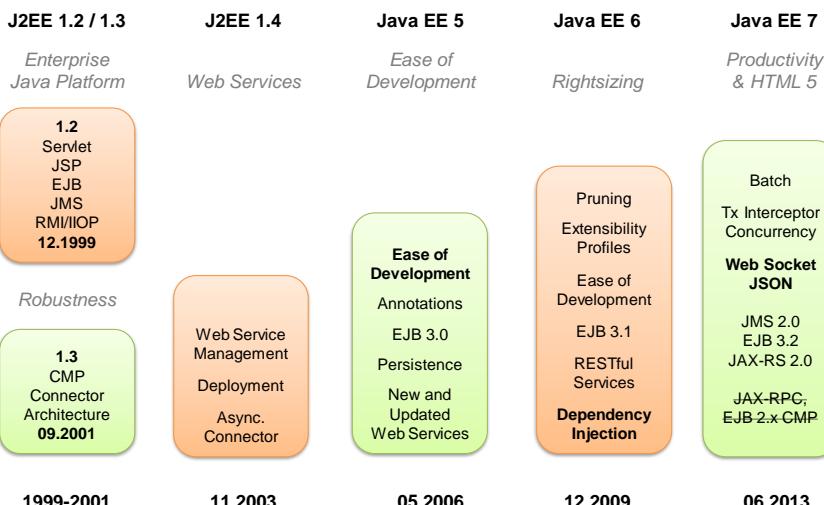
Copyright © iten-engineering.ch

Java Einführung

27

27

## Plattform Evolution



Copyright © iten-engineering.ch

Java Einführung

28

28

## Plattform Evolution II

Java EE 8	Jakarta EE 8	Jakarta EE 9
<i>Last Release of Oracle</i>	<i>Eclipse Enterprise for Java (EE4J)</i>	<i>Putting Enterprise Java to the Cloud</i>
<b>HTTP 2 support with Servlet 4.0</b> New JSON binding API <b>JAX-RS Server-Sent Events &amp; reactive client API</b> <b>Security API</b> cloud and PaaS CDI async Events	New Process New Licensing New Logo Still usage of javax Namespace GlassFish 5.1	Java SE 11 Removal of deprecated Specs New jakarta Namespace GlassFish 5.x maybe more to come...
09.2017	09.2019	

Copyright © iten-engineering.ch      Java Einführung      29



29

## Java EE5

		
Apache Geronimo-2.1.4	Oracle WebLogic Server 10g R3 Interstage Application Server Enterprise Edition 9.2	
		
IBM WASCE 2.0	IBM WebSphere Application Server v7	JBoss Application Server 5.0 JBoss Enterprise Application Platform 5
		
Apusic Application Server (v5.0)	NEC WebOTX 8.1	Oracle Application Server 11
		
OW2 JOnAS 5.1	SAP NetWeaver 7.1	Sun GlassFish Enterprise Server 9.1
		
TmaxSoft JEUS 6	TongTech Co., Ltd TongWeb Application Server 5.0	GlassFish Application Server v2

Copyright © iten-engineering.ch      Java Einführung      30

30

## Java EE6



Copyright © iten-engineering.ch

Java Einführung

31

31

## Java EE7



Copyright © iten-engineering.ch

Java Einführung

32

32

## Java EE8

Java EE 8 Full Platform Compatible Implementations



GlassFish Server Open Source Edition 5.0

Tested Configuration



WildFly 14.x

Tested Configuration



Eclipse GlassFish 5.1

Tested Configuration



IBM WebSphere Application Server Liberty Version 18.0.2

Tested Configuration

RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM 7

Red Hat JBoss Enterprise Application Platform 7.2

Tested Configuration



Java EE 8 Web Profile Compatible Implementations



WildFly 14.x Web Profile

Tested Configuration



IBM WebSphere Application Server Liberty Version 18.0.2

Tested Configuration



Eclipse GlassFish 5.1

Tested Configuration

Stand Q4/2019, aktuelle Liste siehe unter:  
<http://www.oracle.com/technetwork/java/javasee/overview/compatibility-jsp-136984.html>

Copyright © iten-engineering.ch

Java Einführung

33

33

## Jakarta EE 8 – Full Plattform



Apusic AAS  
Kingdee Apusic cloud computing  
Co., Ltd.

Version: 10.1

[Download](#)

Proof of compatibility



Eclipse Glassfish  
Eclipse Foundation

Version: 5.1.0, Full Profile

[Download](#)

Proof of compatibility



JBoss Enterprise  
Application Platform  
Red Hat

Version: 7.3.0.GA

[Download](#)

Proof of compatibility



JEUS  
TmaxSoft Co., Ltd

Version: 8.5

[Download](#)

Proof of compatibility



Open Liberty  
IBM Corporation

Version: 19.0.0.6

[Download](#)

Proof of compatibility



Payara Server  
Payara Services Limited

Version: 5.193.1

[Download](#)

Proof of compatibility



Primeton AppServer  
Primeton

Version: 7

[Download](#)

Proof of compatibility



WildFly  
Red Hat

Version: 18.0.0.Final

[Download](#)

Proof of compatibility

Stand Q3/2020, aktuelle Liste siehe unter: <https://jakarta.ee/compatibility>

Copyright © iten-engineering.ch

Java Einführung

34

34

## Jakarta EE 8 – Web Profile

 <p>Eclipse Glassfish Eclipse Foundation</p> <p>Version: 5.1.0, Web Profile</p> <p><a href="#">Download</a></p> <p><a href="#">Proof of compatibility</a></p>	 <p>JBoss Enterprise Application Platform Red Hat</p> <p>Version: 7.3.0.GA</p> <p><a href="#">Download</a></p> <p><a href="#">Proof of compatibility</a></p>	 <p>Open Liberty IBM Corporation</p> <p>Version: 19.0.0.6</p> <p><a href="#">Download</a></p> <p><a href="#">Proof of compatibility</a></p>	 <p>WildFly Red Hat</p> <p>Version: 18.0.0.Final</p> <p><a href="#">Download</a></p> <p><a href="#">Proof of compatibility</a></p>
--	---	--	---

Stand Q3/2020, aktuelle Liste siehe unter: <https://jakarta.ee/compatibility>

---

Copyright © iten-engineering.ch      Java Einführung      35

35

## Abschnitt IV

### Fazit

---

Copyright © iten-engineering.ch      Java Einführung      36

36

## Herausforderungen

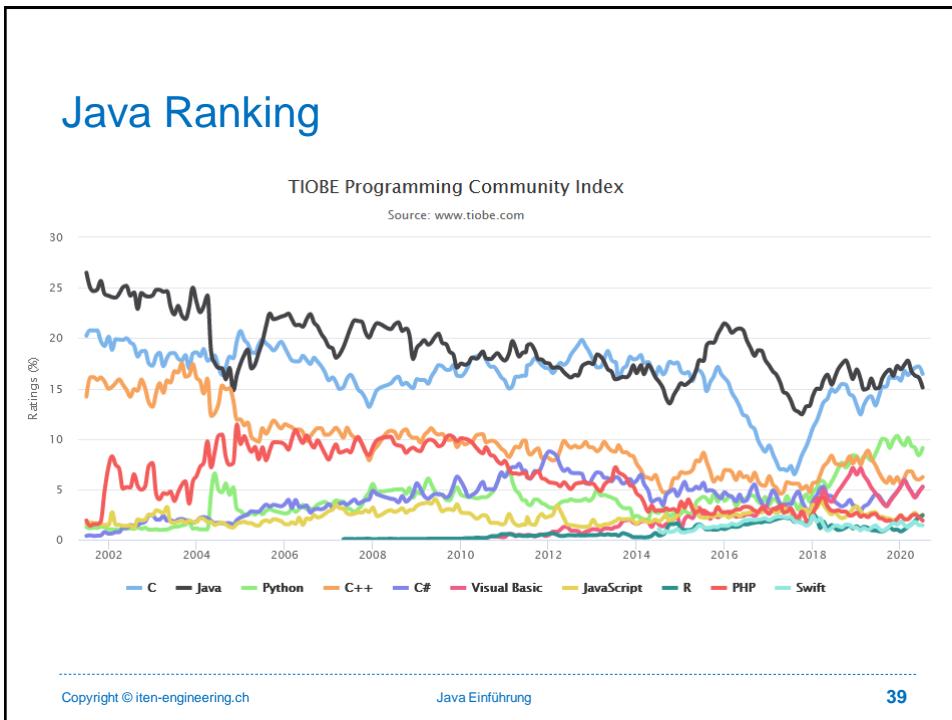
Die heutige Software Entwicklung stellt u.a. folgende Anforderungen:

- ▶ Schnelle und kurze Entwicklungszyklen
- ▶ Steigende Anforderungen an Probabilität und rasche Änderbarkeit
- ▶ Komplexität der Technologien steigt (XML, Java, .Net, Web Services, SOAP, AJAX, etc.)
- ▶ Heterogene Systemlandschaften
- ▶ Zunehmende Wichtigkeit von mobilen Geräten

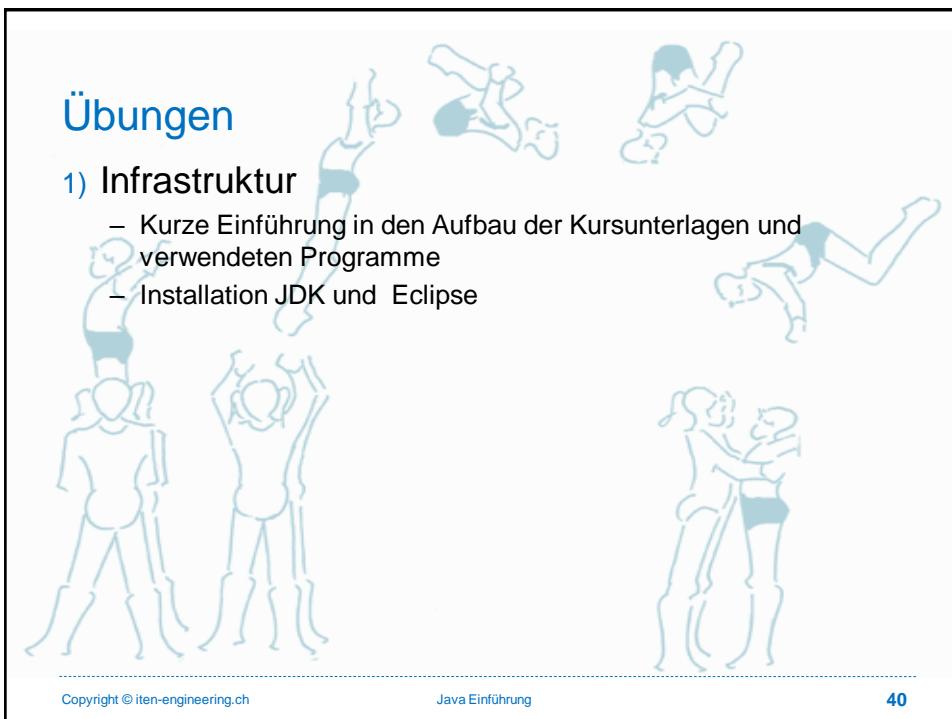
## Vorteile der Java Technologie

Der Einsatz von Java bietet u.a. folgende Vorteile:

- ▶ Java stellt verschiedene Technologie Plattformen für unterschiedliche Anwendungen zur Verfügung
- ▶ Die Plattformen sind stabil und ausgereift
- ▶ Sie integrieren eine Vielzahl verschiedener Technologien und stellen diese dem Entwickler zur Verfügung
- ▶ Unterstützung durch viele verschiedene Lieferanten
- ▶ Mit dem Einsatz von Java erhält man Software Architekturen die einfach zu erstellen, benutzen, unterhalten und erweitern sind



39



40

## Kapitel 3

### Hello World

### Demo Hello World

```

package demo.helloworld;
import java.util.Date;

/**
 * Mein erstes Java Programm.
 */
public class HelloWorld {

    public static void main(String[] args) {

        // einfache Ausgabe auf die Konsole
        System.out.println("Hello World");

        // Ausgabe mit Text und Datum/Zeit
        String text = "Hello World, es ist ";
        Date datum = new Date();
        System.out.println(text + datum);

    } // Ende der Methode main
} // Ende der Klasse HelloWorld

```

Java Klassen werden mit Paketen strukturiert

Import einer Library mit vordefinierten Objekten

Kommentar (JavaDoc)

Klasse

Main Methode, wird als erstes ausgeführt

Ausgabe eines fixen Textes auf die Console

Deklaration Zeichenkette und Zuweisen eines Textes

Erstellen eines Objektes von der Library Klasse Date

Text und Datum zusammensetzen und ausgeben

## Anwendung starten

- ▶ Ein Programm, das als eigenständige Applikation ausgeführt werden soll, muss eine sogenannte **main** Methode enthalten
- ▶ Diese dient als **Startpunkt** der Anwendung
- ▶ Bei Ausführen einer Klasse in der Java Virtual Machine (JVM):
  - wird die Klasse von der JVM geladen
  - nach einer main Methode durchsucht
  - und die Anwendung durch Aufruf der main Methode gestartet

## Komandozeilen Argumente

- ▶ Das einzige Argument an main() ist ein **Array von Strings**, der üblicherweise **args** genannt wird
- ```
public static void main(String[] args)
```
- ▶ Die Elemente dieses Arrays sind die Argumente, die in der Kommandozeile nach dem Klassennamen angegeben wurden
  - ▶ Die Länge des Arrays kann über **args.length** abgefragt werden
  - ▶ Werden keine Argumente übergeben, ist der Array leer (und die Länge ist 0)

## Demo Echo

```
public class Echo {

    public static void main(String[] args) {

        for (int i = 0; i < args.length; i++) {
            System.out.println
                ("Argument #" + i + " = " + args[i]);
        }

        System.out.println
            ("Es wurden " + args.length + 
             " Argumente übergeben.");
    }

}
```

Copyright © iten-engineering.ch

Java Einführung

45

45

## Anwendung verlassen

- ▶ Die main Methode gibt immer void zurück
  - Aus einem Java-Programm kann also kein Wert zurückgegeben werden, indem eine return-Anweisung verwendet wird
- ▶ Wenn ein Wert zurückgeliefert werden soll, wird **System.exit(n)** gemacht
  - wobei n dem gewünschten Integer-Wert entspricht

```
public static void main(String[] args) {
    ...
    System.exit(0);
}
```

- ▶ Die Verarbeitung und Interpretation dieses Exit Wertes hängt vom Betriebssystem ab

Copyright © iten-engineering.ch

Java Einführung

46

46

## Übungen

- 1) HelloJava mit Eclipse
- 2) HelloJava selber kompilieren und ausführen
- 3) ReverseArgs (optional)

## Kapitel 4

### Grundlegende Sprachelemente

## Abschnitt I

### Allgemein

## Bezeichner

- ▶ Java-Bezeichner (**Identifier**) werden zur Benennung von Variablen, Methoden, Klassen, Interfaces, etc. verwendet
- ▶ Sie bestehen aus einer beliebig langen Folge aus **Buchstaben**, **Ziffern** und den Unicode-Zeichen **'\$'** und **'\_'**
- ▶ Java Bezeichner sind **case-sensitive** und dürfen **nicht mit einer Ziffer** beginnen

```
int minValue;      // legal      int $minValue;      // illegal
int _minValue;    // legal      int $;           // legal
int $_;           // legal      int _;           // legal
int _3;           // legal      int a3;          // legal
int 3a;           // illegal    int !maxValue;    // illegal
int - maxValue;   // illegal    int min-Value;   // illegal
```

## Reservierte Wörter

- Die folgenden Wörter sind reserviert und dürfen **nicht als Bezeichner** verwendet werden:
- Schlüsselwörter

|          |          |            |           |              |
|----------|----------|------------|-----------|--------------|
| abstract | continue | for        | new       | switch       |
| assert   | default  | goto       | package   | synchronized |
| boolean  | do       | if         | 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        |

- Literale (vordefinierte Konstanten):

|       |      |      |
|-------|------|------|
| false | true | null |
|-------|------|------|

## Kommentare

- Drei Arten von Kommentaren:
- Zeilenkommentare:
  - Ein Zeilenkommentar ist genau eine Zeile lang
- Blockkommentare:
  - Können sich über mehrere Zeilen strecken
- Dokumentation (Javadoc)
  - Werden vom Javadoc-Tool verwendet
  - Für die automatische Generierung von Klassen- und Interface-Dokumentationen
  - Beginnen mit “/\*\*” und enden mit “\*/”

```
// line comment

/*
block comment
(verschachtelung
Linienkommentar
möglich)
*/

/**
 * javadoc comment
 *
 */

```

## Javadoc

**Method Detail**

**getReport**

```
Report getReport(Integer ipCtrlID,
                 ch.ipi.esv.util.type.language.ReportLanguage reportLanguage)
```

Get the default report ReportType.defaultReport() for an intellectual property for the given language.

The default Report

**Parameters:**

- ipCtrlID - The ctrl ID for the IP.
- reportLanguage - The language of the report.

**Returns:**

The Report.

```
/*
 * Get the default report <code>ReportType.defaultReport()
 * </code>for an intellectual property for the given
 * language.<br />
 *
 * @param ipCtrlID The ctrl ID for the IP.
 * @param reportLanguage The language of the report.
 * @return The Report.
 */
public Report getReport
    (Integer ipCtrlID, ReportLanguage reportLanguage);
```

Copyright © iten-engineering.ch
Java Einführung
53

53

## Anweisungen

- ▶ Ein Programm besteht aus einer Reihe von **Anweisungen** die jeweils mit einem Semikolon abgeschlossen werden
- ▶ Anweisungen können mit geschweiften Klammern { } in Blöcke gruppiert werden

```
int zahl1, zahl2;
int ergebnis;

{
    zahl1 = Integer.parseInt(args[0]);
    zahl2 = Integer.parseInt(args[1]);
}

ergebnis = zahl1 * zahl2;

System.out.println("Das Produkt ist: " + ergebnis);
```

Copyright © iten-engineering.ch
Java Einführung
54

54

## Variablen

- ▶ Variable:
  - Symbol für einen Speicherbereich
  - Speichert Daten eines bestimmten Datentyps
- ▶ Java ist eine stark typisierte Sprache
  - Jede Variable und jeder Ausdruck muss zur Kompilierzeit an einen bestimmten Datentypen gebunden sein
- ▶ Java unterscheidet zwischen Referenzen (Objekte) und primitiven Datentypen
- ▶ Primitive Datentypen sind keine Objekte
  - können schneller verarbeitet werden (Performance)
  - boolean, char, byte, short, int, long float, double

## Variablen Deklaration

- ▶ Syntax:

Type identifier [, identifier...];

Type identifier = initial value [, ...];

```
int x, y;           // Initialwert ist 0
boolean b;          // Initialwert ist false
String s;           // Initialwert ist null

int maxlen = 500;    // Deklarationen mit
String txt = "Hallo"; // Initialisierung
int x=1, y=2;
```

## Konstanten

- ▶ Konstanten werden in Java mit dem Schlüsselwort `final` deklariert
- ▶ Eine einmal definierte Konstante kann nicht mehr geändert werden
- ▶ Bei Klassen werden die Konstanten zusätzlich mit `static` definiert (so dass nur eine Instanz im System vorhanden ist)

```
final int z1;      // Konstante innerhalb Methode
z1 = 500;         // gültig
z1=555;          // ungültig

// Konstante in einer Klasse
public static final int MAX_GEWICHT = 2500;
```

## Methoden

- ▶ Eine Methode besteht aus der Deklaration und einem Methodenkörper, wobei dieser in geschweiften Klammern steht.
  - Public bedeutet, dass die Methode für alle sichtbar ist (aufgerufen werden kann)
  - Als Parameter werden die beiden Variablen a und b von Typ int übergeben
  - Das Resultat der Methode ist vom Typ int

```
public int add (int a, int b) {
    int resultat = a + b;

    return resultat;
}
```

## Methoden II

- Wenn eine Methode keine Rückgabewert hat, wird dies mit `void` angegeben

```
public void print(int x, int y) {
    System.out.println("x=" + x);
    System.out.println("y=" + y);
}
```

- Falls keine Parameter benötigt werden, verwendet man ein leeres Klammerpaar `()`

```
public String getFirstname() {
    return this.firstname;
}
```

## Scope

- Der Gültigkeitsbereich einer Variablen wird als **Scope** bezeichnet
- Innerhalb des Scope darf ein Variablenname nur einmal vorkommen
- Wenn der Scope endet, kann auf die Variable nicht mehr zugegriffen werden
- Variablen innerhalb eines Anweisungsblokkes einer Methode werden als **lokale Variablen** bezeichnet

## Beispiel Scope

```
public class Scope {  
  
    public static int z = 1000;           // globale Variable  
   // definieren  
  
    public static void print() {  
        int x = 3;                      // lokale Variable  
        System.out.println("x=" + x);     // Ausgabe: x=3  
        System.out.println("z=" + z);     // Ausgabe: z=1000  
    }  
  
    public static void main(String[] args) {  
        int x = 100;                    // lokale Variable  
  
        System.out.println("x=" + x);     // Ausgabe: x=100  
        System.out.println("z=" + z);     // Ausgabe: z=1000  
    }  
}
```

## Abschnitt II

### Datentypen

## boolean

- ▶ Logische Variablen haben zwei Zustände, die man als **an oder aus, wahr oder falsch, ja oder nein** interpretieren kann
- ▶ In Java wird ein solcher Wahrheitswert durch den Datentyp **boolean** repräsentiert
- ▶ Sein Wertebereich besteht nur aus zwei Werten, nämlich **true und false**

```
boolean isFirst = true;
boolean hasFailed = false;
```

## char

- ▶ 16 Bit Unicode-Zeichen bzw. Ganzzahlen aus dem Intervall  $[0, 2^{16}-1]$
- ▶ *char* ist der einzige numerische Datentyp, der nicht vorzeichenbehaftet ist
- ▶ Mit *char*-Variablen können ganzzahlige Berechnungen durchgeführt werden

```
char a ='a';                      // entspricht der Zahl 97
char b ='b';                      // entspricht der Zahl 98
char c =99;
int res1 = a+b;
int res2 = res1+c;

System.out.println      // Ausgabe: a+b=195, a+b+c=294
    ("a+b=" + res1 + ",a+b+c=" + res2);
```

## byte, short, int, long

- **Ganzzahltypen** können in 3 Formaten dargestellt werden:
  - Dezimalformat : z.B. 281
  - Oktalzahlen (Zahlen zur Basis 8 ):  
beginnen mit „0“ z.B. 0431
  - Hexadezimal (Zahlen zur Basis 16 ):  
beginnen mit „0x“ z.B. 0x119

```

int _281_as_decimal      = 281;
int _281_as_oktal        = 0431;
int _281_as_hexadecimal = 0x119;

System.out.println          // Ausgabe: 281 281 281
(_281_as_decimal+" "+_281_as_oktal+" "+_281_as_hexadecimal);

```

Copyright © iten-engineering.ch

Java Einführung

65

65

## float, double

- Stellen **Fließkommazahlen** dar
- Double sind doppelt so lang und doppelt so genau wie Float Zahlen

```

float _22_divided_by_7_as_float   = 22 / 7.f;
double _22_divided_by_7_as_double = 22 / 7.d;

// Ausgabe:
// _22_divided_by_7_as_float: 3.142857
// _22_divided_by_7_as_double: 3.142857142857143

System.out.println
(" _22_divided_by_7_as_float:" + _22_divided_by_7_as_float);

System.out.println
(" _22_divided_by_7_as_double:" + _22_divided_by_7_as_double);

```

Copyright © iten-engineering.ch

Java Einführung

66

66

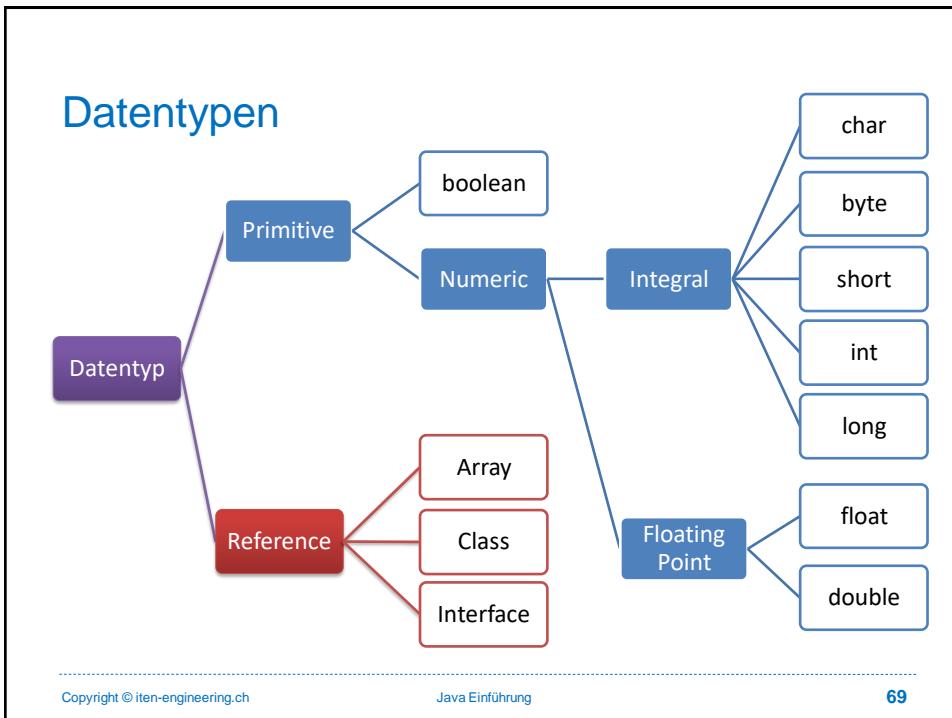
## float, double II

- ▶ float werden mit dem Suffix „f“ oder „F“ gekennzeichnet
  - Beispiel: 2.4f oder 2.4F
- ▶ Double haben den Suffix „d“ oder „D“
  - Beispiel: 2.4d oder 2.4D
- ▶ Eine **suffixlose** Kommazahl wird vom Compiler automatisch als **Double-Zahl** interpretiert
  - 2.4 ist eine Double-Zahl
- ▶ Um eine Fließkommazahl von einer Ganzzahl unterscheiden zu können, muss mindestens der **Dezimalpunkt**, der **Exponent** oder der **Suffix** vorhanden sein
  - Beispiele: 1., .1, 2E4, 2e4, 2f, 5D

## Wrapper Klassen

- ▶ Zu jedem primitiven Datentypen in Java gibt es eine korrespondierende Wrapper-Klasse
- ▶ Diese kapseln die primitiven Variable in einer objektorientierten Hülle
- ▶ Wrapper-Klassen verfügen über zahlreiche Methoden und Konstanten
- ▶ Beispiele:

| Primitiver Typ | Wrapper Klasse | Konstanten                      |
|----------------|----------------|---------------------------------|
| boolean        | Boolean        | FALSE, TRUE                     |
| char           | Character      | MIN_VALUE, MAX_VALUE, SIZE      |
| byte           | Byte           | MIN_VALUE, MAX_VALUE, SIZE, NaN |



69

## Datentypen II

| Typ                       | Grösse [Bit]       | Beispiel                      | Default  | Minimum            | Maximum                                 | Wrapper Class |
|---------------------------|--------------------|-------------------------------|----------|--------------------|-----------------------------------------|---------------|
| byte                      | 8                  | byte b = 65;                  | 0        | -128               | 127                                     | Byte          |
| char                      | 16                 | char c = 'A';<br>char c = 65; | '\u0000' | '\u0000'           | '\uffff' <sup>(1)</sup>                 | Character     |
| short                     | 16                 | short s = 65;                 | 0        | -2 <sup>15</sup>   | 2 <sup>15</sup> -1                      | Short         |
| int                       | 32                 | int i = 65;                   | 0        | -2 <sup>31</sup>   | 2 <sup>31</sup> -1                      | Integer       |
| long                      | 64                 | long l = 65L;                 | 0        | -2 <sup>63</sup>   | 2 <sup>63</sup> -1                      | Long          |
| float                     | 32                 | float f = 65f;                | 0.0f     | 2 <sup>-149</sup>  | (2-2 <sup>-23</sup> )·2 <sup>127</sup>  | Float         |
| double                    | 64                 | double d = 65.55;             | 0.0d     | 2 <sup>-1074</sup> | (2-2 <sup>-52</sup> )·2 <sup>1023</sup> | Double        |
| boolean                   | 1                  | boolean b = true;             | false    | n/a                | n/a                                     | Boolean       |
| String<br>(or any Object) | n/a <sup>(2)</sup> | String t = "Hello"            | null     | n/a                | n/a                                     | n/a           |

<sup>(1)</sup> 16 Bit Unicode Zeichenbereich  
<sup>(2)</sup> Die Grösse für die Reference selber ist ein Implementationsdetail der Java Virtual Machine.

Copyright © iten-engineering.ch      Java Einführung      70

70

## Typenkompatibilität und Typenkonversion

- ▶ Jede Variable in Java ist an einen Typ gebunden
- ▶ Java erlaubt den Datentyp einer Variable in einen anderen Datentyp zu konvertieren
- ▶ Zwei Arten von Typkonvertierungen:
- ▶ **Implizite** (automatische ) Typkonvertierung:  
Wird vom Compiler automatisch durchgeführt
- ▶ **Explizite** Typumwandlung (casting):  
Wird im Programm explizit angegeben

## Beispiele implizite Typenkonvertierung

```

int x = 5 / 2;           // Ganzzahl Division: x ist vom
                        // Typ int und hat den Wert 2!

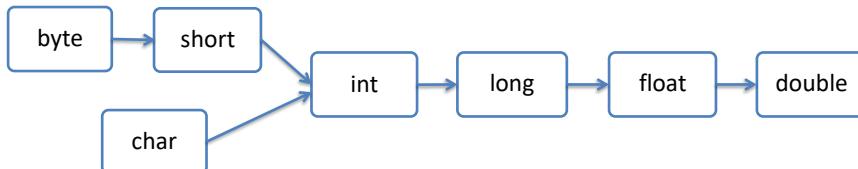
double y = 5 / 2.0;      // Fliesskomma Division, y ist vom
                        // Typ double und hat den Wert 2.5

float z = 5 / 2.0f;      // Fliesskomma Division, z ist vom
                        // Typ float und hat den Wert 2.5

```

## Casting

- ▶ Ob explizit oder implizit **hängt** vom **Ziel-Typ** ab
- ▶ Jede Typkonvertierung entgegen der Pfeilrichtung muss explizit durchgeführt werden
- ▶ Explizite Typkonvertierung wird als **Casting** bezeichnet (einschränkende Konvertierung)
- ▶ Konvertierungen in Pfeilrichtung werden automatisch durchgeführt (erweiternde Typkonvertierungen)



## Beispiele Casting

```

short s = 4;
char c = 'g';
int i = 32;

double d = i;           // erweiternde Typumwandlung

float f = (float) d;    // Cast (einschränkende
                       // Typumwandlung)

byte b = (byte) c;      // Cast

c = (char) s;          // Cast

c = (char) b;          // Cast
  
```

## Autoboxing (boxing/unboxing)

- ▶ Autoboxing gibt es erst seit Java Version 5
- ▶ Vor Version 5 waren die primitiven Datentypen und ihre korrespondierenden Wrapper-Klassen nicht zuweisungskompatibel
- ▶ So war es z. B. nicht möglich eine primitive Integer-Variable einem Integer-Objekt oder umgekehrt zuzuweisen:

```
Integer x = 2;                      // Ungültig

int y = new Integer(3);              // Ungültig
```

## Autoboxing II

- ▶ Unter **Boxing** versteht man die automatische Umwandlung eines primitiven Wertes in ein Wrapper-Klassenobjekt
- ▶ **Unboxing** ist die Umkehrung von Boxing, sprich die automatische Umwandlung eines Wrapper-Klassenobjektes in einen primitiven Wert

```
Boolean b = false;                  // boxing

float f = new Float(3.4f);          // unboxing
```

## Autoboxing III

- ▶ Da die Umwandlung **automatisch** vom Compiler erfolgt, spricht man von Autoboxing
- ▶ Primitive Datentypen und ihre entsprechenden Wrapper-Klassen sind zuweisungskompatibel
- ▶ Vor Version 5 hat musste man die Umwandlung manuelle durchführen

```
// Vor Version 5 manuelle Umwandlung
```

```
Boolean b = new Boolean(false);

Float f1= new Float(3.4f);
float f = f1.floatValue();
```

Copyright © iten-engineering.ch

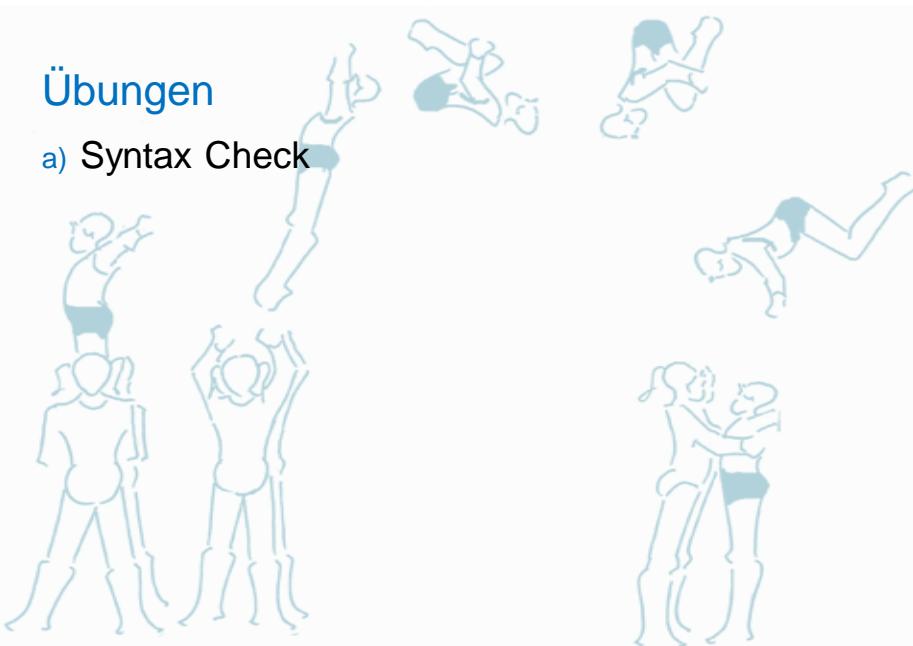
Java Einführung

77

77

## Übungen

- a) Syntax Check



Copyright © iten-engineering.ch

Java Einführung

78

78

## Abschnitt III

### Operatoren

### Arithmetische Operatoren

- ▶ Operatoren werden nur auf primitive Datentypen (und Strings [+]) angewendet

- + Addition
- Subtraktion; Vorzeichen
- \* Multiplikation
- / Division (int-Typen: ganzzahliger Anteil:  $17/3 = 5$ )
- % Divisionsrest (nur bei int-Typen:  $17\%3 = 2$ )

## Arithmetische Operatoren II

```

int x = 8;
int y = 4;
int resultat;

resultat = x + y;           // resultat = 12
System.out.println(x + " + " + y + " = " + resultat);

resultat = x - y;           // resultat = 4
System.out.println(x + " - " + y + " = " + resultat);

resultat = x * y;           // resultat = 32
System.out.println(x + " * " + y + " = " + resultat);

resultat = x / y;           // resultat = 2
System.out.println(x + " / " + y + " = " + resultat);

```

## Reduced Notation & Precedence

```

int x = 5;
int y = 3;

x = x + 1;           // increment
x++;

x = x - 1;           // decrement
x--;

x = x + y;           // reduced notation
x += y;

x = x - y;           // reduced notation
x -= y;

int res = ((x * 5) + y) / 2;    // rangfolge (precedence)
                                // mit Klammern

```

## Prefix und Postfix

- ▶ Bei der reduzierten Notation wird zwischen **Pre-** und **Postfix** unterschieden
- ▶ Im ersten Fall findet die Operation **vor** der Zuweisung statt, im zweiten Fall **nach** der Zuweisung

```

int i = 10;
int j = 10;
int resultat;

resultat = 2 * ++i;      // i = 11, resultat = 22

resultat = 2 * j++;      // resultat = 20, j = 11

```

## Logische Operatoren

- ▶ Logische Operatoren liefern als Ergebnis **true** oder **false**
- ▶ Dies kann durch einen **Vergleich** zustande kommen oder durch **logische Verknüpfungen**
- ▶ Nachfolgende Tabelle zeigt das Ergebnis logischer Verknüpfungen mit AND, OR und XOR

| x | y | AND | OR | XOR |
|---|---|-----|----|-----|
| 0 | 0 | 0   | 0  | 0   |
| 0 | 1 | 0   | 1  | 1   |
| 1 | 0 | 0   | 1  | 1   |
| 1 | 1 | 1   | 1  | 0   |

## Logische Operatoren II

```
// AND &&
System.out.println("AND");
System.out.println("false && false = " + (false && false));
System.out.println("true && false = " + (true && false));
System.out.println("false && true = " + (false && true));
System.out.println("true && true = " + (true && true));

// OR ||
System.out.println("OR");
System.out.println("false || false = " + (false || false));
System.out.println("true || false = " + (true || false));
System.out.println("false || true = " + (false || true));
System.out.println("true || true = " + (true || true));

// XOR ^
System.out.println("XOR");
System.out.println("false ^ false = " + (false ^ false));
System.out.println("true ^ false = " + (true ^ false));
System.out.println("false ^ true = " + (false ^ true));
System.out.println("true ^ true = " + (true ^ true));
```

## Vergleichsoperatoren

- ▶ Mit Vergleichsoperatoren werden Ausdrücke formuliert, die als Resultat eine boolean liefern
- ▶ In Java können alle mathematischen Vergleiche mithilfe einfacher Zeichen dargestellt werden:

**==** gleich (Achtung, nicht verwechseln mit der **=** Zuweisung)  
**!=** nicht gleich  
**>=** grösser als oder gleich  
**<=** kleiner als oder gleich  
**>** grösser als  
**<** kleiner als

- ▶ Angewendet werden diese auf fast alle primitiven Datentypen

## Vergleichsoperatoren II

```
int x = 8;  
int y = 4;  
  
boolean test;  
  
test = (x == y);           // resultat = false  
System.out.println(x + " == " + y + " = " + test);  
  
test = (x > y);          // resultat = true  
System.out.println(x + " > " + y + " = " + test);  
  
test = (x <= y);         // resultat = false  
System.out.println(x + " <= " + y + " = " + test);  
  
test = (x >= y);         // resultat = true  
System.out.println(x + " >= " + y + " = " + test);
```

## Abschnitt IV

### Ein- und Ausgabe

## Ein- und Ausgabe

- ▶ Generell erfolgt in Java die Ein- und Ausgabe von Daten mit sogenannten **Streams**
- ▶ Diese werden wir in einem späteren Kapitel betrachten
- ▶ Für einfache **Ausgaben** auf die **Konsole** stehen einige einfache Methoden der Klasse **System** zur Verfügung
- ▶ Für die **Eingabe** von Daten via **Konsole** steht im Projekt die Hilfsklasse **Reader** zur Verfügung

## Daten ausgeben

- ▶ Für einfache Ausgabe kann mit den beiden System Methoden **System.out.print()** und **System.out.println()** gearbeitet werden
  - Mit **println** wird neben der Ausgabe noch zusätzlich ein **Zeilenvorschub** ausgeführt
- ▶ Für formatierte Ausgaben mit Text und Parametern eignet sich die Methode **System.out.printf()** sehr gut
  - Hier kann ein Text mit Platzhaltern versehen werden, an deren Stelle dann die mitgelieferten Variablen automatisch eingesetzt werden

## Daten ausgeben II

- Auswahl Platzhalter für printf:

%s = String  
 %d = Ganzzahliger Wert  
 %g = Fliesskomma Wert  
 %n = Zeilenumbruch  
 %% = Ausgabe des % Zeichen selber

```

// Ausgabe eines String
String message = Reader.readln("String = ");
System.out.printf("Sie haben die Meldung <%s> eingegeben.%n", message);

// Ausgabe von Zahlen
int min = 12;
int max = 48;

System.out.printf
  ("Der gültige Eingabebereich liegt zwischen %d und %d.%n", min, max);
  
```

Copyright © iten-engineering.ch

Java Einführung

91

91

## Daten einlesen

- Für das Einlesen von `int`, `double` und `Strings` via Konsole kann die Klasse `Reader` aus dem Package `util` verwendet werden
- Dabei gibt es jeweils eine Methode die den Wert einliest und eine, die vor dem Einlesen noch einen Text (Prompt) ausgibt

```

System.out.println("Reader Demo:");

String input = Reader.readln("String = ");
System.out.println(input);

int x = Reader.readInt("int = ");
System.out.println(x);

double y = Reader.readDouble("double = ");
System.out.println(y);
  
```

Copyright © iten-engineering.ch

Java Einführung

92

92

## Daten konvertieren

- ▶ Für die Konvertierung von Daten (z.B. Strings) in einen anderen Typen, bieten die [Wrapper Klassen](#) entsprechende [parse Methoden](#)
- ▶ Damit können zum Beispiel Programm Parameter (welche dem Hauptprogramm als Strings übergeben werden) in einen anderen Typen konvertiert werden
- ▶ Damit die Konvertierung möglich ist, muss die Eingabe ein [gültiges Format](#) aufweisen
- ▶ Eine detaillierte Beschreibung zu den einzelnen Funktionen findet man im Javadoc der jeweiligen Wrapper Klassen

## Parameter konvertieren

```
public static void main(String[] args) {

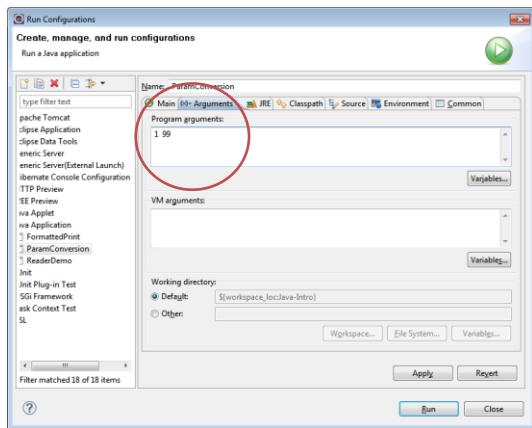
    // check params
    if (args.length != 2) {
        System.out.println
            ("Bitte übergeben Sie dem Programm zwei
             Ganzzahl Parameter (min / max).");
        System.exit(-1);
    }

    // read and convert params
    int min = Integer.parseInt(args[0]);
    int max = Integer.parseInt(args[1]);

    System.out.printf("Parameter min=%d, max=%d", min, max);
    System.exit(0);
}
```

## Übergabe von Parametern mit Eclipse

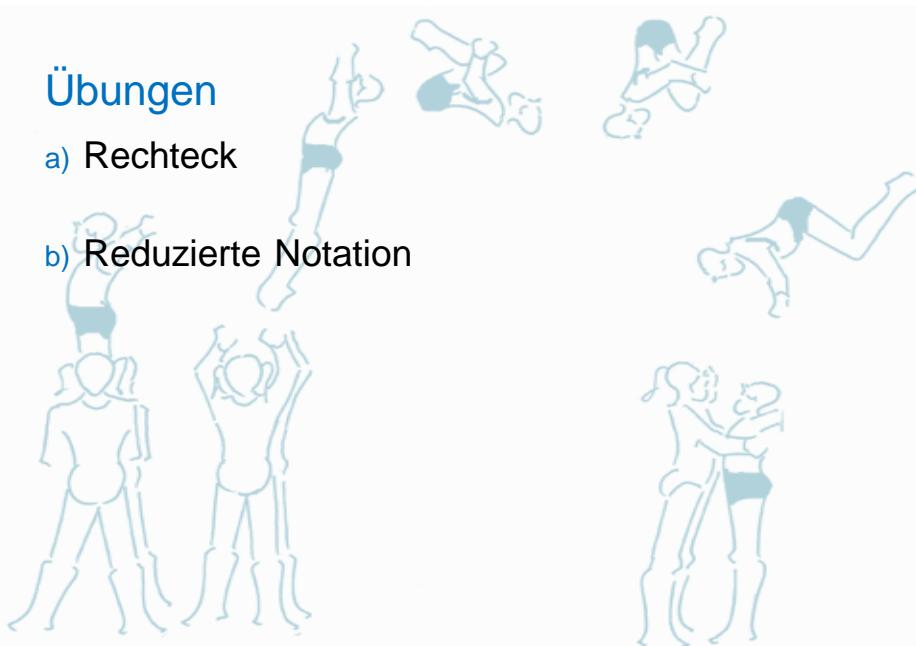
- ▶ Selektieren Sie Ihr Anwendung
- ▶ Wählen Sie anschliessend auf den Start Button den Eintrag «Run Configurations...»
- ▶ Geben Sie im Register Arguments die Parameter ein



## Übungen

a) Rechteck

b) Reduzierte Notation



## Kapitel 5

### Kontrollstrukturen

### Sequenz

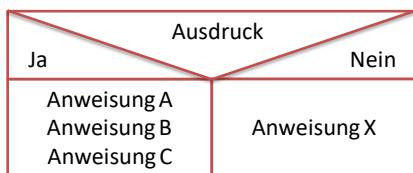
```
public static void main(String[] args) {  
    int zahl1, zahl2;  
    int ergebnis;  
  
    zahl1 = Integer.parseInt(args[0]);  
    zahl2 = Integer.parseInt(args[1]);  
  
    ergebnis = zahl1 * zahl2;  
  
    System.out.println("Das Produkt ist: " + ergebnis);  
}
```

Einlesen zahl1  
Einlesen zahl2  
Ergebnis =  
zahl1 \* zahl2  
Ausgabe Ergebnis

## Verzweigung

```
// if (einseitige Verzweigung)
if (x > 100) {
    System.out.println("x ist grösser 100");
}

// if - else (zweiseitige Verzweigung)
if (x > 100) {
    System.out.println("x ist grösser 100");
} else {
    System.out.println("x ist kleiner/gleich 100");
}
```



Copyright © item-engineering.ch

Java Einführung

99

99

## Geschachtelte Verzweigung

```
// if - else (geschachtelte Verzweigung)
if (x > 100) {
    System.out.println("x ist grösser 100");
} else {
    if (x > 50) {
        System.out.println("x ist zwischen 51..100");
    } else {
        System.out.println("x ist kleiner/gleich 50");
    }
}
```

Copyright © item-engineering.ch

Java Einführung

100

100

## Mehrseitige Verzweigung

```
// if - else if (mehrseitige Verzweigung)
if (x > 100) {

    System.out.println("x ist grösser 100");

} else if (x > 50) {

    System.out.println("x ist zwischen 51..100");

} else if (x > 25) {

    System.out.println("x ist zwischen 26..50");

} else {

    System.out.println("x ist kleiner/gleich 50");
}
```

Copyright © iten-engineering.ch

Java Einführung

101

101

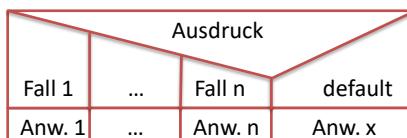
## Fallauswahl

```
// switch (Fallauswahl)
int note = 5;
switch (note) {
    case 6:
        System.out.println
            ("ausgezeichnet");
        break;

    case 5:
        System.out.println("gut");
        break;

    case 4:
        System.out.println("genügend");
        break;

    default:
        System.out.println("ungenügend oder ungültig");
        break;
}
```



Seit Java 7 können auch Strings im case verwendet werden!

Copyright © iten-engineering.ch

Java Einführung

102

102

## Schleifen

```
// while (Kopfgesteuerte Schleife)
int z = 5;

while (z > 0) {
    System.out.println(z);

    z--;
}
```

while Bedingung

Anweisung(en)

```
// do (Fussgesteuerte Schleife)
int y;

do {
    y = Reader.readInt("y=");
    System.out.println(y);

} while (y >= 0);
```

do

Anweisung(en)

while Bedingung

Copyright © iten-engineering.ch

Java Einführung

103

103

## Zählergesteuerte Schleife

```
// for (Zählergesteuerte Schleife)

for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

Initialisierung Statement

Aktualisierung Statement

Bedingung

int i = 1

i <= 10

println (i)

i++

Copyright © iten-engineering.ch

Java Einführung

104

104

## Abbruch / Weiterfahren von Schleifen

```
// break (Abbruch Schleife)
for (int i = 1; i <= 10; i++) {
    System.out.println(i);

    if (i == 5) {
        // schlaufe wird verlassen / beendet
        break;
    }
}

// continue (Weiterfahren mit nächsten Lauf der Schleife)
for (int i = 1; i <= 10; i++) { ←

    if (i == 5) {
        // restliche Anweisungen werden übersprungen
        continue; ←
    }
    System.out.println(i);
}
```

Copyright © iten-engineering.ch

Java Einführung

105

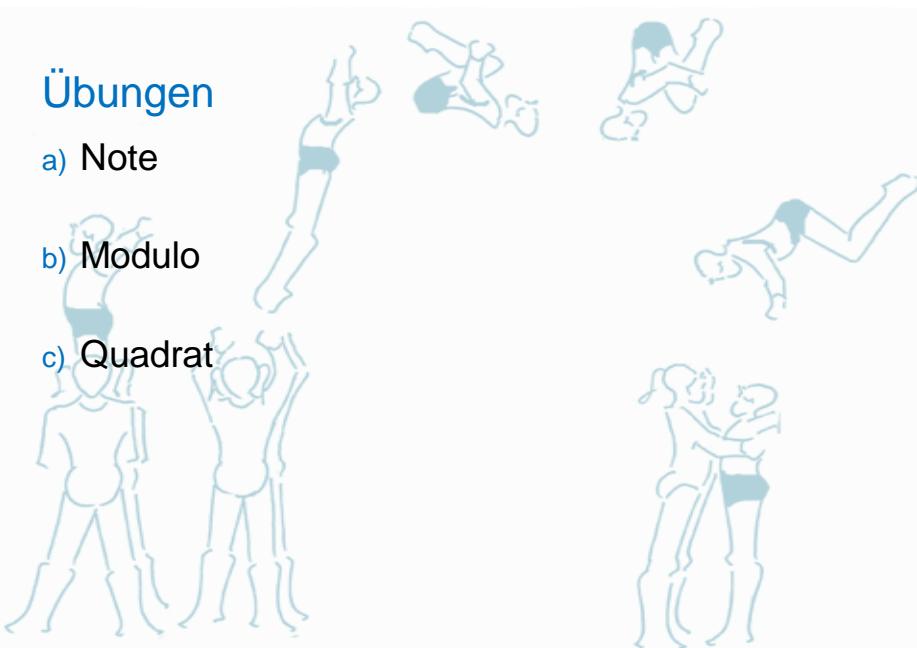
105

## Übungen

a) Note

b) Modulo

c) Quadrat



Copyright © iten-engineering.ch

Java Einführung

106

106

## Kapitel 6

### Klassen, Attribute und Methoden

---

Copyright © iten-engineering.ch

Java Einführung

107

107

## Abschnitt I

### Prinzipien der Objektorientierung

---

Copyright © iten-engineering.ch

Java Einführung

108

108

## Reale Gegenstände sind komplex

- ▶ Die realen Gegenstände und deren Beziehungen zu einander sind sehr komplex und können nicht ohne weiteres in einem Softwaresystem abgebildet werden.
- ▶ Um diese Dilemma zu beheben, wird folgender Lösungsansatz verfolgt:
  - das Problem auf das wesentliche zu reduzieren
  - Modularisierung
  - strukturierte Einordnung (Hierarchie)
  - Kapselung zusammengehöriger Informationen

## Prinzipien

- ▶ Mit der Objektorientierung wird nun versucht, Methoden und Konzepte anzubieten, die dem menschlichen Denken nachempfunden sind.
- ▶ Daraus ergeben sich folgende Prinzipien der Objektorientierung:

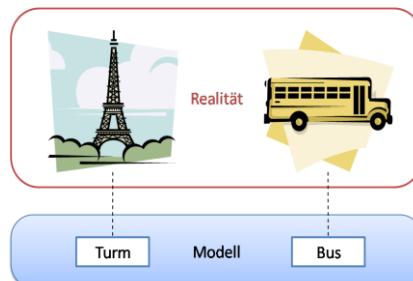


## Abstraktion

- ▶ Der Mensch bedient sich des Prinzips Abstraktion, um mit Komplexität umgehen zu können.
- ▶ Abstraktion bedeutet:
  - sich auf das Wesentliche zu konzentrieren, also Unwichtiges ausseracht zu lassen
  - und Gemeinsamkeiten zwischen verschiedenen Objekten zu erkennen.
- ▶ Die Umwelt ist von sich aus komplex. Um sie einfacher verstehen zu können, werden diejenigen Informationen bewusst ausgewählt, die benötigt werden, um einen Sachverhalt verständlich zu machen.
- ▶ Dabei wird in Kauf genommen (und sogar absichtlich darauf abgezielt), dass ein Teil der Realität nicht berücksichtigt wird.

## Vereinfachtes Modell der Realität

- ▶ Durch die Anwendung von Abstraktion erhalten wir also ein Modell der Realität.
- ▶ Dadurch werden Objekte identifiziert sowie deren Beziehungen, Abhängigkeiten und Verhalten modelliert.
- ▶ Es findet dabei eine Trennung zwischen dem Konzept (Klassen) Umsetzung (Instanzen) statt.



## Grad der Abstraktion

- ▶ Eine Klasse beschreibt folgende Dinge:
  - Beschreibung der Vorgehensweise zum Erzeugen einer neuen Instanz (Konstruktor)
  - gemeinsame Attribute
  - gemeinsame (interne) Verhaltensweisen (private Methoden)
  - gemeinsame Reaktionen (Verhaltensweisen nach Nachrichtenempfang, öffentliche Methoden)
- ▶ Die Schwierigkeit bei der Definition von Klassen besteht darin, den Grad der Abstraktion entsprechend dem Anwendungsfall geeignet zu wählen.

## Kapselung

- ▶ Durch das Prinzip Kapselung wird jedes Objekt zu einer abgeschlossenen Einheit, die über definierte Schnittstellen mit der Umwelt kommunizieren kann.
- ▶ Beispiel: Black Box mit Knöpfen, Schaltern und Anzeigefeldern.
- ▶ Diese Form der Modularisierung hilft dabei, Problemlösungen durch Lösung von Teilproblemen zu erzielen; dies bedeutet also eine Vereinfachung der Lösung.

## Kapselung II



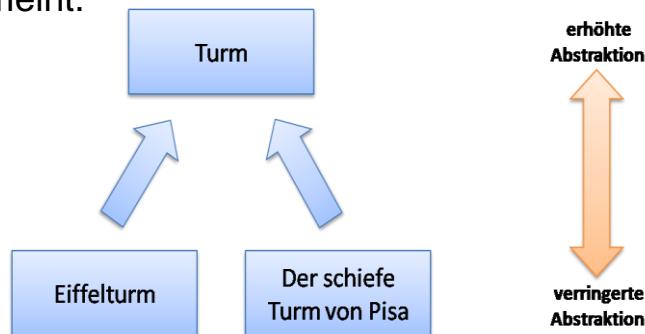
- ▶ Kapselung bedeutet, dass Attribute (Eigenschaften) und Methoden (Verhaltensweisen) in einem Objekt zusammengefasst werden.
- ▶ Dabei wird zwischen **Innen- und Aussensicht** unterschieden.

## Vererbung

- ▶ Bei der Betrachtung von Klassenhierarchien werden
  - alle Vorgänger einer Klasse als Oberklassen / Superklassen
  - und alle Nachfolger als Unterklassen / Subklassen bezeichnet.
- ▶ Eine Klasse erbt alle Eigenschaften der unmittelbaren Oberklasse und gibt all ihre Eigenschaften an die unmittelbare Unterklasse weiter.
- ▶ Geerbte Eigenschaften werden dabei ebenfalls weitergegeben.

## Vererbung II

- Dieser Vorgang wird als Vererbung bezeichnet.
- Mit Eigenschaften sind dabei Methoden und Attribute der jeweils betrachteten Klasse gemeint.



Copyright © iten-engineering.ch

Java Einführung

117

117

## Polymorphismus (Vielgestaltigkeit)

- Zur menschlichen Denkweise gehört es, kontextabhängig zu denken, d.h.
  - Nachrichten also entsprechend ihrem jeweiligen Kontext einzuordnen.
- In der OO, die versucht, die menschliche Denkweise abzubilden
  - daher können Methoden kontextabhängig verwendet werden.
- Diese Fähigkeit wird in der Programmierung als Polymorphismus bezeichnet.
- Wörtlich übersetzt bedeutet Polymorphie »Vielgestaltigkeit«.

Copyright © iten-engineering.ch

Java Einführung

118

118

## Beispiel Fortbewegungsmittel

- ▶ Nehmen wir als Beispiel eine Oberklasse Fortbewegungsmittel, die eine Methode bewege() besitzt.
- ▶ Fortbewegungsmittel hat die Unterklassen Auto, Flugzeug und Schiff.
- ▶ Jede dieser Klassen implementiert die Methode bewege().
- ▶ Trifft nun die Nachricht „bewege“ bei einem dieser Objekte ein (d.h. die entsprechende Methode wird aufgerufen), reagiert jedes Objekt unterschiedlich.
- ▶ Das Auto fährt auf dem Land, das Flugzeug fliegt in der Luft, und das Schiff bewegt sich auf dem Wasser



## Abschnitt II

### Klassen, Attribute und Methoden

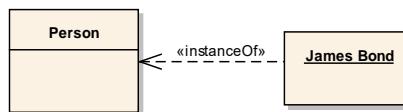
## Klassen

- ▶ Eine Klasse **definiert** die Gemeinsamkeiten (Eigenschaften und Operationen) von Objekten
- ▶ Eine Klasse dient als Vorlage für Objekte, die jeweils unterschiedliche Eigenschaften (Attributwerte) haben können
- ▶ Konvention Klassen:
  - Verwende Substantiv als Name für Klassen
  - Verwende Einzahl für Klassename
  - Schreibe ersten Buchstaben gross
  - Schreibe ersten Buchstaben eines Teilworts gross



## Objekte / Instanzen

- ▶ Jede Instanz einer Klasse ist ein Objekt, das individuell angesprochen und manipuliert werden kann
- ▶ Jedes Objekt hat
  - eine Identität
  - einen Zustand  
(Menge der Attributwerte zu einem bestimmten Zeitpunkt)
  - ein Verhalten (Operationen)
  - Beziehungen zu anderen Objekten
- ▶ Konvention Objekte
  - Schreibe Objektnamen klein
  - Schreibe ersten Buchstaben eines Teilworts gross



## Attribute

- ▶ Objekte haben Eigenschaften, die von aussen erfragt werden können.
- ▶ Dabei kann von aussen nicht unterschieden werden, ob eine Eigenschaft direkt auf Daten des Objekts basiert oder ob die Eigenschaft auf der Grundlage von Daten berechnet wird.
- ▶ Eigenschaften, die nicht direkt auf Daten basieren, werden abgeleitete Eigenschaften genannt.
- ▶ Konventionen Attribute:
  - Verwende Substantiv als Name für Attribute
  - Verwende ggf. vorangestelltes Adjektiv
  - Schreibe ersten Buchstaben klein
  - Schreibe ersten Buchstaben eines Teilwortes gross

## Beispiel Klasse mit Attributen

```
public class Rectangle1 {                                // Klasse Rectangle1
    int length = 0;                                    // mit zwei Attributten
    int width = 0;                                     // für die Länge und Breite
}

public class Rectangle1Demo {

    public static void main(String[] args) {
        // Instanz rect von Klasse Rectangle1 erzeugen
        Rectangle1 rect = new Rectangle1();

        // Attribute setzen
        rect.length = 10;
        rect.width = 5;

        // Attribute lesen und ausgeben
        System.out.println
            ("Rechteck mit Länge " + rect.length +
             " und Breite " + rect.width);
    }
}
```

## Operationen und Methoden

- ▶ Operationen spezifizieren, welche Funktionalität ein Objekt bereitstellt.
- ▶ Unterstützt ein Objekt eine bestimmte Operation, sichert es einem Aufrufer damit zu, dass es bei einem Aufruf die Operation ausführen wird.
- ▶ Durch die Operation wird dabei zum einen die Syntax des Aufrufs vorgegeben, also zum Beispiel für welche Parameter Werte eines bestimmten Typs zusammen mit dem Aufruf übergeben werden müssen.
- ▶ Methoden von Objekten sind die konkreten Umsetzungen von Operationen.
  - Während Operationen die Funktionalität nur abstrakt definieren, sind Methoden für die Realisierung dieser Funktionalität zuständig.
- ▶ Konventionen Operationen:
  - Verwende Verben als Name für Methoden
  - Schreibe ersten Buchstaben eines Teilwortes gross
  - Schreibe Parameter klein

## Beispiel Methoden

```
public class Rectangle2 {
    int width = 0;
    int length = 0;

    // einfache Methode
    void swapWidthLength() {
        int temp = width;           // lokale Variable
        width = length;
        length = temp;
    }

    // Methode mit einem Parameter
    void buildSquare(int sideLength) {
        width = sideLength;
        length = sideLength;
    }

    // Methode ohne Parameter und mit Rueckgabewert
    int getArea() {
        return width * length;
    }
}
```

## Beispiel Methoden überladen

```
public class Rectangle2 {

    int width = 0;
    int length = 0;

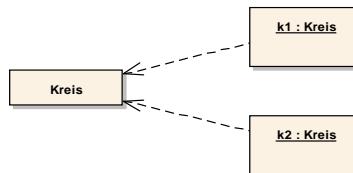
    ...

    // Methode ueberladen
    void resize(int newWidth, int newLength) {
        width = newWidth;
        length = newLength;
    }

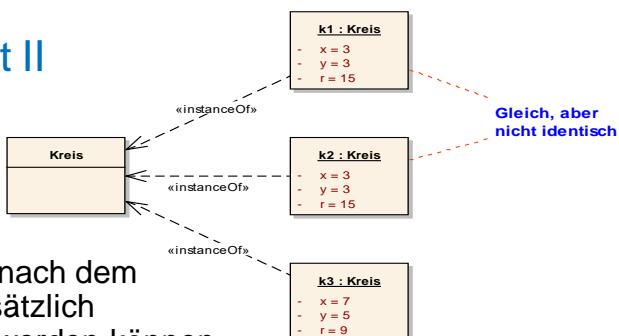
    // Zweite Methode resize mit anderer Signatur
    void resize(double factor) {
        width = (int) (width * factor);
        length = (int) (length * factor);
    }
}
```

## Objektidentität

- ▶ Die Identität eines Objektes ist nicht zu verwechseln mit seinen Attributwerten.
- ▶ Jede Objekt Identität ist per Definition unabhängig von seinen konkreten Attributwerten.
- ▶ Zwei Objekte können dadurch immer unterschieden werden, auch wenn sie zu einem Zeitpunkt exakt die gleichen Attributwerte aufweisen.



## Objektidentität II



- ▶ Das Kriterium, nach dem Objekte grundsätzlich unterschieden werden können, wird Identitätskriterium genannt.
- ▶ In vielen Programmiersprachen ist die Adresse des Objekts im Speicherbereich das generell verfügbare Identitätskriterium.
- ▶ In objektorientierten Anwendungen können auch andere Kriterien verwendet werden, wie zum Beispiel die Übereinstimmung einer eindeutigen Kennung.

## Beispiel Objektidentität

```

public class Rectangle3Demo {

    public static void main(String[] args) {

        Rectangle3 rectA = new Rectangle3();
        rectA.length = 10;
        rectA.width = 5;

        Rectangle3 rectB = new Rectangle3();
        rectB.length = 10;
        rectB.width = 5;

        System.out.println("Rechteck A mit Länge " + rectA.length
            + " und Breite " + rectA.width + " und Id " + rectA);
        System.out.println("Rechteck B mit Länge " + rectB.length
            + " und Breite " + rectB.width + " und Id " + rectB);
    }
}
  
```

## Statische Variablen

- ▶ Die Attribute einer Klasse gehören genau zu einer Instanz der Klasse
  - Dass heisst auch, sie existieren erst, wenn eine solche Instanz angelegt wird
- ▶ Im Gegensatz dazu gehören statische Variablen zur Klasse selbst
  - Sie werden daher auch als Klassenvariablen bezeichnet
  - Der Zugriff erfolgt über die Klasse selbst, d.h. es ist keine Instanz notwendig
  - Statische Variablen existieren genau einmal (in der ganzen Anwendung)

## Statische Methoden

- ▶ Es ist auch möglich, statische Methoden zu erstellen
- ▶ Diese sind ebenfalls unabhängig von einer bestimmten Instanz und werden über die Klasse referenziert
- ▶ Alle Methoden und Attribute der Klasse `java.lang.Math` (welche math. Grundfunktionen definiert) sind zum Beispiel statisch

```
sqrt
public static double sqrt(double a)

Returns the correctly rounded positive square root of a double value. Special cases:
• If the argument is NaN or less than zero, then the result is NaN.
• If the argument is positive infinity, then the result is positive infinity.
• If the argument is positive zero or negative zero, then the result is the same as the argument.
Otherwise, the result is the double value closest to the true mathematical square root of the argument value.

Parameters:
    a - a value.

Returns:
    the positive square root of a. If the argument is NaN or less than zero, the result is NaN.
```

Ausschnitt  
Javadoc von  
`java.lang.Math`

## Beispiel statische Variablen und Methoden

```

class StaticElements {
    static int DotsPerInch = 300;

    static double calcSquare(double wert) {
        return wert * wert;
    }
}

public class StaticElementsDemo {
    public static void main(String[] args) {
        System.out.printf ("Die Druckerauflösung ist %d dpi %n",
                           StaticElements.DotsPerInch);

        double value = 5.0;
        double result = StaticElements.calcSquare(value);

        System.out.printf ("Das Quadrat von %g ist %g %n", value, result);
    }
}

```

Copyright © iten-engineering.ch

Java Einführung

133

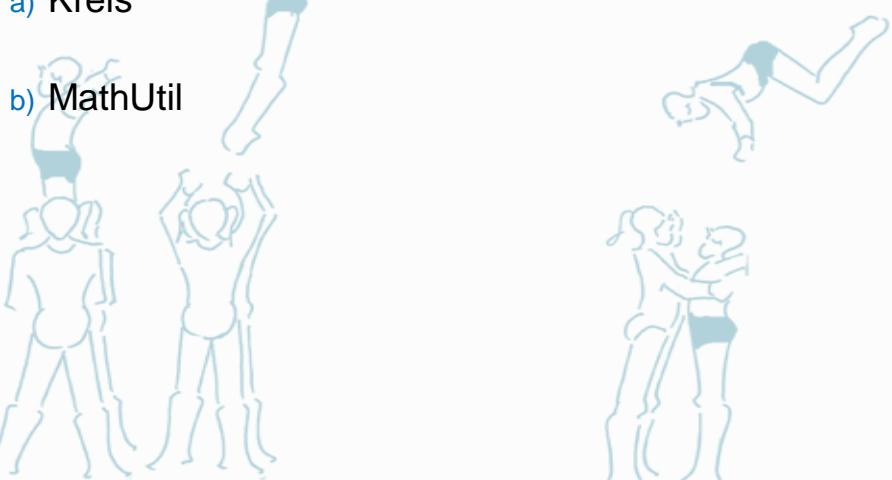
133

## Übungen

a) Kreis



b) MathUtil



Copyright © iten-engineering.ch

Java Einführung

134

134

## Kapitel 7

### Kapselung und Konstruktoren

---

Copyright © iten-engineering.ch

Java Einführung

135

135

## Abschnitt I

### Kapselung

---

Copyright © iten-engineering.ch

Java Einführung

136

136

## Kapselung



- ▶ Kapselung bedeutet, dass Attribute (Eigenschaften) und Methoden (Verhaltensweisen) in einem Objekt zusammengefasst werden.
- ▶ Dabei wird zwischen **Innen- und Aussensicht** unterschieden.

## Innen- und Aussensicht

- ▶ Innensicht
  - Der interne Aufbau und das interne Verhalten eines Objektes werden nach außen nicht gezeigt.
  - Man spricht in diesem Zusammenhang vom Geheimnisprinzip (engl. information hiding).
- ▶ Aussensicht
  - Von außen sind nur die zugehörigen Methodennamen eines Objektes sichtbar, aber nicht ihre konkrete Implementation (Programmcode).
- ▶ Ein Objekt stellt klar definierte Schnittstellen zur Verfügung, ohne die internen Details der Implementation preiszugeben

## Vorteile

- ▶ Ein Vorteil der Kapselung von Attributen und Methoden ist, dass **nur erlaubte Zugriffe möglich** sind
- ▶ Bei der Arbeit mit einem Objekt ist somit ein **unbeabsichtigter Zugriff auf interne Informationen nicht möglich**, da für diese Daten keine Schnittstelle nach außen definiert ist
- ▶ Des weiteren kann die **interne Struktur eines Objektes beliebig verändert** werden kann, ohne dass Programme, die dieses Objekt nutzen, umgestellt werden müssen
- ▶ Anders ausgedrückt ist die **Verwendung eines Objektes unabhängig** von dessen **Implementierung**

## Zugriffsrechte

- ▶ Zur Unterstützung der Datenkapselung definiert Java sogenannte **Modifier**  
`public, private, protected, final,  
abstract, static, native, transient,  
synchronized, volatile`
- ▶ Diese teilen dem Compiler Informationen mit, wie z.B. Sichtbarkeit, Veränderbarkeit und Lebensdauer von Klassen, Variablen, Konstruktoren, Methoden und Initialisierer
- ▶ Die Modifier **public**, **private** und **protected** bezeichnet man als **Zugriffsmodifikatoren (Access-Modifier)**
  - Dazu kommt noch die Variante, wenn keine Angaben gemacht werden, dies bezeichnet man als **default**

## Access Modifier

- ▶ Access Modifier können bei einer Klasse bei den Variablen, Konstruktoren, Methoden oder der Klasse selbst deklariert werden
- ▶ Mit dem Modifier **private** versehene Attribute und Methoden sind nur innerhalb der Klasse sichtbar

```
public class Rectangle4 {
    private int width = 0; // Attribut mit dem Modifier private
    private int length = 0; // sind nur innerhalb der Klasse sichtbar
    ...
}
```

## Beispiel mit Getter und Setter

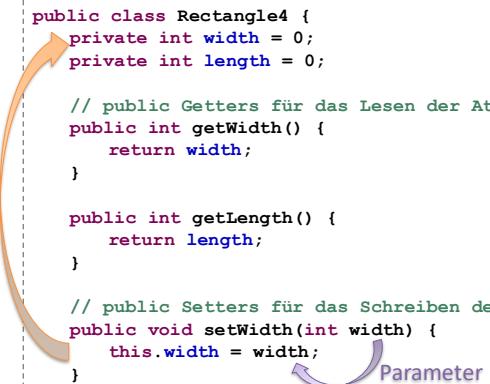
```
public class Rectangle4 {
    private int width = 0;
    private int length = 0;

    // public Getters für das Lesen der Attribute von aussen
    public int getWidth() {
        return width;
    }

    public int getLength() {
        return length;
    }

    // public Setters für das Schreiben der Attribute von aussen
    public void setWidth(int width) {
        this.width = width;
    }

    public void setLength(int length) {
        this.length = length;
    }
}
```



Die Referenzvariable **this** ermöglicht den Zugriff auf die Attribute und Methoden innerhalb des Objektes

## Beispiel mit Getter und Setter II

```
public class Rectangle4Demo {

    public static void main(String[] args) {

        // Erstellung Objekt via Standard Konstruktor
        Rectangle4 rect = new Rectangle4();

        // Attribute setzen via Setter
        rect.setLength(10);
        rect.setWidth(5);

        // Attribute lesen via Getter
        System.out.println("Rechteck mit Länge " + rect.getLength()
            + " und Breite " + rect.getWidth());
    }
}
```

## Verantwortlichkeiten (Kohärenzprinzip)

- ▶ Jede Klasse soll genau für einen fachlichen Aspekt verantwortlich sein soll.
- ▶ In diesem Zusammenhang ist das Kohärenzprinzip wichtig, dass besagt, dass zusammengehörende Verantwortlichkeiten nicht geteilt, sondern in einer Klasse konzentriert werden sollen.
- ▶ Die für diese fachliche Verantwortlichkeit zuständigen Eigenschaften sollen somit in einer einzelnen Klasse zusammengefasst werden.
- ▶ Auf der anderen Seite, soll eine Klasse keine Eigenschaften enthalten, die nicht zum entsprechenden Verantwortlichkeitsbereich gehören

## Demeter Gesetz

- ▶ Das Gesetz von Demeter (Law of Demeter) besagt, dass Objekte nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren sollen.
- ▶ Dadurch soll die Kopplung unter den Objekten verringert und Aufrufketten vermieden werden.
- ▶ Bei Klassen die eine zentrale Rolle einnehmen und dadurch viele Eigenschaften anderer Fachklassen benötigen, sollen die Abhängigkeiten durch sogenannte Controller Klassen entkoppelt werden

## Abschnitt II

### Konstruktoren

## Konstruktoren

- ▶ Mit Konstruktoren werden konkrete Instanzen / Objekte einer Klasse erstellt
- ▶ Für jede Klasse die keine eigenen Konstruktoren definiert, stellt Java automatisch einen **Standard Konstruktor** zur Verfügung:

```
// Erstellung Objekt via Standard Konstruktor
Rectangle4 rect = new Rectangle4();
```

- ▶ In vielen Fällen ist es praktisch, eigene Konstruktoren zu erstellen
  - Dabei kann eine Klasse auch **mehrere** verschiedene Konstruktoren definieren
  - Innerhalb eines Konstruktor ist es auch möglich einen anderen Konstruktor aufzurufen (Konstruktoren **verketten**)

## Beispiel mit Konstruktoren

```
public class Rectangle5 {
    private int width = 0;
    private int length = 0;

    // Standard Konstruktor ohne Parameter
    public Rectangle5() {
        this.width = 1;      // Alternativ könnte auch der zweite
        this.length = 1;     // Konstruktor mit this(1,1) aufgerufen werden
    }

    // Konstruktor mit Breite und Länge
    public Rectangle5(int width, int length) {
        this.width = width;
        this.length = length;
    }

    // Konstruktor mit gleicher Seitengröße für Länge und Breite (Quadrat)
    public Rectangle5(int size) {
        this(size, size); // Aufruf des anderen Konstruktors
    }
    ...
}
```

## Beispiel mit Konstruktoren II

```
public class Rectangle5Demo {
    public static void main(String[] args) {
        // Ein Rechteck mit der Breite 1 und der Laenge 1 erzeugen
        Rectangle5 specialRect = new Rectangle5();

        // Ein Rechteck mit der Breite 3 und der Laenge 5 erzeugen
        Rectangle5 anotherRect = new Rectangle5(3, 5);

        System.out.printf("Das Rechteck (%d x %d)%n",
                           specialRect.getWidth(), specialRect.getLength());

        System.out.printf("hat den Flaecheninhalt %d%n",
                           specialRect.getArea());
        ...
    }
}
```

Copyright © iten-engineering.ch

Java Einführung

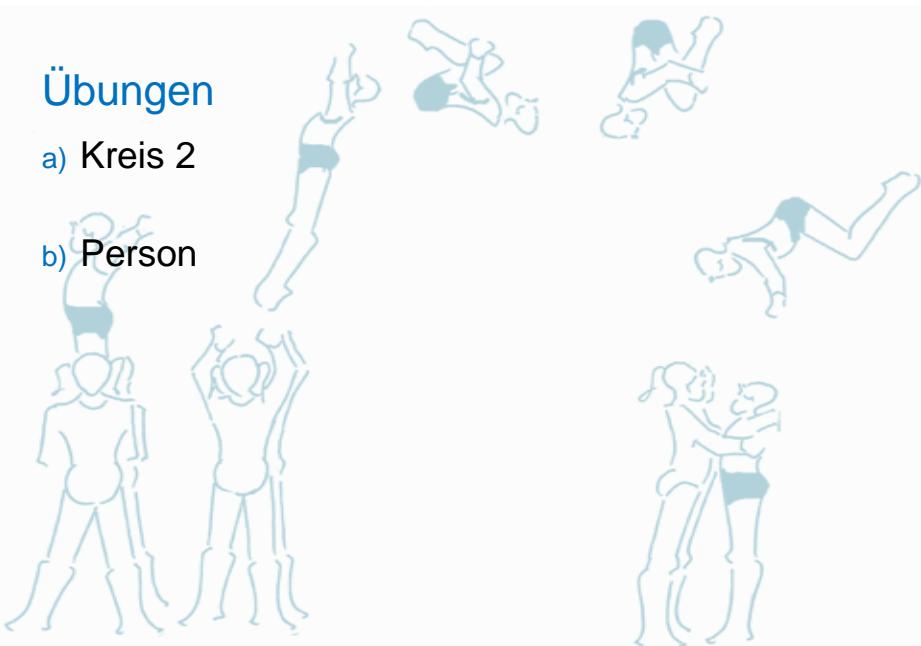
149

149

## Übungen

a) Kreis 2

b) Person



Copyright © iten-engineering.ch

Java Einführung

150

150

## Kapitel 8

### Vererbung

151

## Abschnitt I

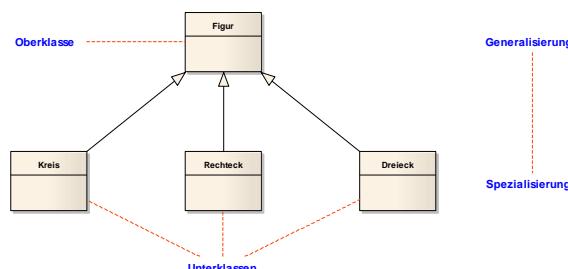
### Grundlagen

152

## Vererbungsprinzip

- ▶ Klassen können hierarchisch angeordnet werden
- ▶ Die übergeordnete Klasse bezeichnet man als **Oberklasse** (engl. **Superclass**) und die abgeleiteten Klassen als **Unterklassen** (engl. **Subclass**)
- ▶ Dabei erben die abgeleiteten Klassen die Eigenschaften und Operationen der ihnen übergeordneten Klassen
  - sofern diese durch die Sichtbarkeitsregeln für die Nutzung in Unterklassen freigegeben sind

## Spezialisierung und Generalisierung



- ▶ Die Klassen Kreis, Rechteck und Dreieck sind jeweils eine **Spezialisierung** der Klasse Figur
  - Unterklassen erben die in den Oberklassen bereits implementierte Funktionalität
  - Diese Funktionalität kann unverändert übernommen oder in Teilen von den Unterklassen überschrieben (spezialisiert) werden
  - Es ist auch möglich, neue Eigenschaften und Operationen hinzuzufügen
  - Bestehende Eigenschaften sollen jedoch nicht unterdrückt oder beschränkt werden!
- ▶ Umgekehrt stellt die Klasse Figur eine **Generalisierung** der genannten Unterklassen dar

## Vererbung mit Java

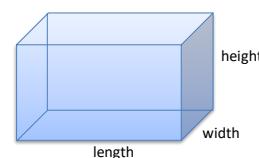
- ▶ In Java besitzt eine Klasse genau eine unmittelbare Oberklasse
- ▶ Ist keine konkrete Oberklasse angegeben, wird **standardmäßig Objekt** als Oberklasse festgelegt
- ▶ Durch diese Strukturierung lassen sich eindeutige Pfade innerhalb einer Klassenhierarchie bilden
- ▶ Jede Ebene in der Hierarchie stellt eine Abstraktionsebene dar
- ▶ Im Hinblick auf Wiederverwendbarkeit ist es wichtig, dass eine Klasse der jeweiligen Hierarchieebene nur Methoden und Attribute enthält, die diese Ebene allgemein beschreiben

## Eine Klasse ableiten und Erweitern

- ▶ Zu einer Klasse kann eine Oberklasse angegeben werden.
- ▶ Dies geschieht durch das Schlüsselwort **extends**
- ▶ Beispiel mit Rechteck und Quader:

```
class Rectangle {
    private int width = 0;
    private int length = 0;
    ...
}

class Cuboid extends Rectangle {
    private int height;
    ...
}
```



## Zugriff auf Oberklasse

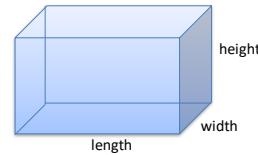
- ▶ Für den Zugriff auf die Oberklasse stehen alle mit **public** oder **protected** gekennzeichneten Attribute und Methoden zur Verfügung
- ▶ Da in unserem Beispiel die Attribute **private** sind erfolgt der Zugriff via **Getter und Setter**

```
class Cuboid extends Rectangle {
    private int height;

    ...

    public int getVolume() {
        return this.getArea() * this.height;
    }
}
```

Methode von Rectangle



Copyright © iten-engineering.ch

Java Einführung

157

157

## Konstruktoren

- ▶ Bei der Vererbung werden nur Methoden und Attribute vererbt aber **keine Konstruktoren**
- ▶ Wird in einer Subklasse kein Konstruktor implementiert, so steht einzig der **Default – Konstruktor** zur Verfügung
  - Dieser ruft seinerseits den parameterlosen Konstruktor der Oberklasse auf
- ▶ Um eigene Initialisierungen durchführen zu können, werden in der Subklasse i.d.R. **eigene Konstruktoren** definiert
  - Diese können bei Bedarf auch die Konstruktoren der Oberklasse benutzen / aufrufen

Copyright © iten-engineering.ch

Java Einführung

158

158

## Beispiel Konstruktoren

```

class Rectangle {
    private int width = 0;
    private int length = 0;

    // Standard Konstruktor ohne Parameter
    public Rectangle() {
        this.width = 1;
        this.length = 1;
    }
    ...
}

class Cuboid extends Rectangle {
    private int height;

    // Standard Konstruktor
    Cuboid() {
        this.height = 1;
    }

    ...
}

```



automatischer Aufruf  
des Standard Konstruktor  
von Rectangle

Copyright © iten-engineering.ch

Java Einführung

159

159

## Beispiel Konstruktoren II

```

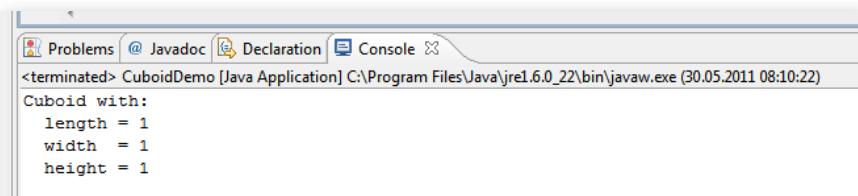
public class CuboidDemo {

    public static void main(String[] args) {

        Cuboid c = new Cuboid();

        System.out.println("Cuboid with: ");
        System.out.println("  length = " + c.getLength());
        System.out.println("  width  = " + c.getWidth());
        System.out.println("  height = " + c.getHeight());
    }
}

```



Copyright © iten-engineering.ch

Java Einführung

160

160

## Konstruktoren mit super

- ▶ Verwendet man Konstruktoren der Oberklasse, so werden diese über das Schlüsselwort **super** aufgerufen
- ▶ Der Aufruf muss dabei an **erster Stelle** im eigenen Konstruktor stehen
- ▶ Falls kein Konstruktor der Oberklasse aufgerufen wird,
  - so wird als erste Anweisung
  - automatisch der Standard Konstruktor der Oberklasse ausgeführt (falls vorhanden)

## Beispiel Konstruktoren mit super

```
class Rectangle {
    private int width = 0;
    private int length = 0;

    // Konstruktor mit Breite und Länge
    public Rectangle(int width, int length) {
        this.width = width;
        this.length = length;
    }
    ...
}

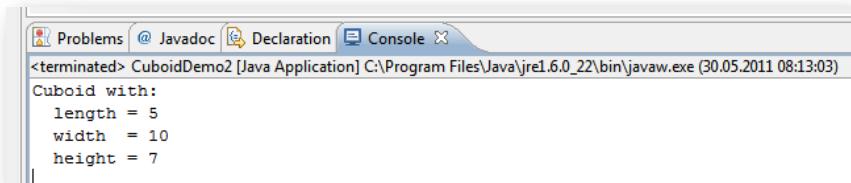
class Cuboid extends Rectangle {
    private int height;

    Cuboid(int width, int length, int height) {
        super(width, length);
        this.height = height;
    }
    ...
}
```

Konstruktor der Superklasse (Oberklasse)  
wird aufgerufen

## Beispiel Konstruktoren mit super II

```
public class CuboidDemo2 {
    public static void main(String[] args) {
        Cuboid c = new Cuboid(10, 5, 7);
        System.out.println("Cuboid with: ");
        System.out.println("  length = " + c.getLength());
        System.out.println("  width  = " + c.getWidth());
        System.out.println("  height = " + c.getHeight());
    }
}
```



## Attribute

- ▶ Alle Attribute, die nicht als private in der Oberklasse deklariert sind, werden vererbt. d.h. man kann direkt auf sie zugreifen
- ▶ Vererbte Attribute können „überschrieben“ (Overriding) werden, indem Attribute gleichen Namens in der Subklasse deklariert werden

## Methoden

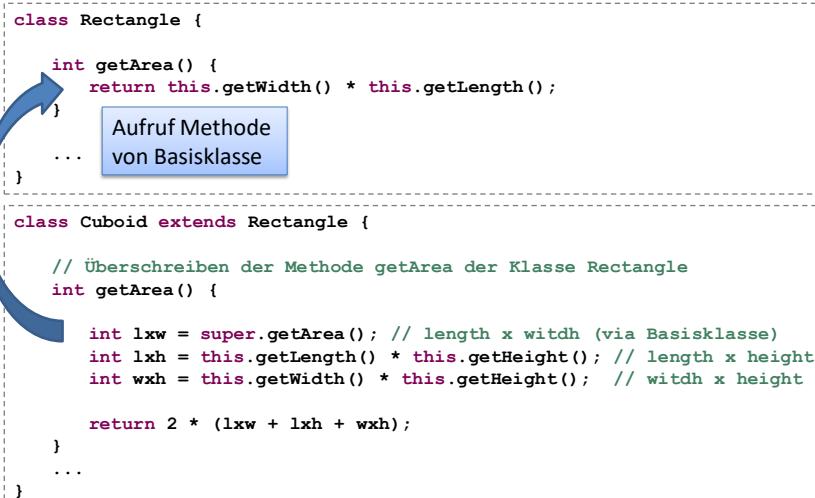
- ▶ Alle Methode, die nicht als private in der Oberklasse deklariert sind, werden vererbt
- ▶ Vererbte Methoden können „überschrieben“ werden, indem Methoden gleicher Signatur (Name und Parameter) deklariert werden
- ▶ Methoden gleichen Namens mit anders lautender Signatur bestehen danach weiterhin

## Beispiel Methoden

```
class Rectangle {
    int getArea() {
        return this.getWidth() * this.getLength();
    }
    ...
}

class Cuboid extends Rectangle {
    // Überschreiben der Methode getArea der Klasse Rectangle
    int getArea() {
        int lwx = super.getArea(); // length x width (via Basisklasse)
        int lhx = this.getLength() * this.getHeight(); // length x height
        int wxh = this.getWidth() * this.getHeight(); // width x height

        return 2 * (lwx + lhx + wxh);
    }
    ...
}
```

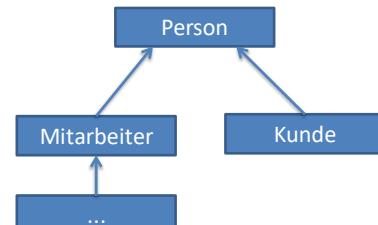


## Abschnitt II

### Vererbungshierarchie

### Vererbungshierarchie

- ▶ Von einer Subklasse können weitere Klassen abgeleitet werden
- ▶ Damit ist es möglich, ganze Hierarchien abzubilden



```
public class Person {  
    private String name;  
    private int jahrgang;  
}  
  
public class Mitarbeiter extends Person {  
    private int lohn;  
}  
  
public class Kunde extends Person {  
    private int kundenNr;  
}
```

## Polymorphie

- ▶ Innerhalb einer Vererbungshierarchie können Methoden mit dem **gleichen Namen** oder gar der **gleichen Signatur** auftreten.
  - Im Rahmen der Vererbung werden „kontextabhängige Methoden“ i. d. R. weit oben in der Klassenhierarchie angesiedelt, oft in so genannten „abstrakten Klassen“
  - Die konkrete Ausprägung einer Methode wird dann weiter unten in den jeweiligen konkreten Klassen individuell realisiert (durch „Überladen“ der Methoden)
- ▶ Es wird dennoch sichergestellt, dass jeweils die „richtige“ Methode aufgerufen wird
- ▶ In Java existieren mit **early** und **late Binding** zwei Ausprägungen von Polymorphismus

## Beispiel Polymorphie

```
public class Person {
    private String name;
    private int jahrgang;

    public String toString() {
        return this.name + " hat Jahrgang " + this.jahrgang;
    }
}

public class Mitarbeiter extends Person {
    private int lohn;

    public String toString() {
        return super.toString() + " mit Lohn " + this.lohn;
    }
}

public class Kunde extends Person {
    private int kundenNr;
    public String toString() {
        return super.toString() + " mit KundenNr " + this.kundenNr;
    }
}
```

## Early Binding

- ▶ Einfache Form von Polymorphismus, wird auch als Überladen ([engl. Overloading](#)) bezeichnet.
- ▶ Dabei haben verschiedene Methoden den gleichen Namen und den gleichen Rückgabewert, aber unterschiedliche Signaturen.
- ▶ Eine Methoden Signatur besteht dabei aus drei Bestandteilen:
  - Methodename
  - Datentypen der Argumentliste und deren Reihenfolge
  - Rückgabewert (wird nicht berücksichtigt)
- ▶ Beim [early binding](#) wird die richtige Methode schon beim Übersetzen des Codes herausgefunden, d.h. zur [Build / Compile Time](#).

## Late Binding

- ▶ Wird auch als Überschreiben ([engl. Overriding](#)) bezeichnet
- ▶ Tritt auf, wenn in einer UnterkLASSE eine Methode implementiert wird, die dieselbe Signatur (also auch denselben Namen) besitzt wie eine Methode einer übergeordneten Klasse
- ▶ Im Gegensatz zum statischen Binden kann hier [erst zur Laufzeit](#) ([engl. Runtime](#)) entschieden werden, welche Methode aufgerufen werden soll, denn das ist abhängig davon, in welchem Objekt diese Methode angesprochen wird.
- ▶ Beim [Overriding](#) müssen alle derartigen Methoden in den Superklassen gleiche, oder schwächere Zugriffsbeschränkungen haben, wie in den Subklassen.

## Substitutionsprinzip

- ▶ Bei den Beziehungen der Ober- und Unterklassen spricht man von einer „ist ein“ Semantik“.
- ▶ In unserem Beispiel mit den Personen gilt:
  - Ein Mitarbeiter ist eine Person
  - Ein Kunde ist eine Person
- ▶ Es gilt das Prinzip, dass Objekte von abgeleiteten Klassen jederzeit anstelle von Objekten ihrer Basisklasse(n) eingesetzt werden können.
- ▶ Dies bezeichnet man als Substitutionsprinzip

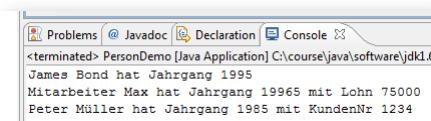
## Beispiel Substitutionsprinzip & Overriding

```
public class PersonDemo {
    public static void main(String[] args) {
        Person person = new Person("James Bond", 1995);
        Mitarbeiter mitarbeiter =
            new Mitarbeiter("Mitarbeiter Max", 19965, 75000);
        Kunde kunde = new Kunde("Peter Müller", 1985, 1234);

        // Liste vom Typ der Oberklasse (Person)
        ArrayList<Person> list = new ArrayList<Person>();

        list.add(person);
        list.add(mitarbeiter);
        list.add(kunde);

        // Ausgabe der Personen
        for (Person p : list) {
            // Anzeige aller Personen
            // mit Hilfe von Overriding (Late Binding)
            System.out.println(p.toString());
        }
    }
}
```



## Kompatibilität

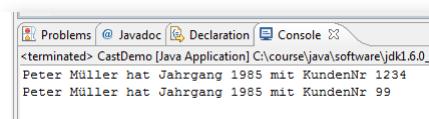
- ▶ Wie das Beispiel zeigt, kann eine abgeleitete Klasse anstelle Ihrer Oberklasse eingesetzt werden
- ▶ Die abgeleiteten Klassen sind kompatibel mit der Oberklasse
- ▶ Sie verfügen über alle Attribute und Methoden der Oberklasse
- ▶ Sie sind somit vollständig als ein solches Objekt funktionstüchtig

## Typenkonversion

- ▶ Wenn einer Referenzvariablen der Oberklasse ein Objekt einer Unterklasse zugewiesen wird, so sind nur noch die Methoden und Attribute der Oberklasse verfügbar
- ▶ Möchte man wieder das ursprüngliche Objekt herstellen, so kann man dies mit einem expliziten **cast** machen

```
public static void main(String[] args) {
    Person person = new Kunde("Peter Müller", 1985, 1234);
    System.out.println(person);

    // cast
    Kunde kunde = (Kunde) person;
    kunde.setKundenNr(99);
    System.out.println(person);
}
```



## Typkompatibilität

- Macht man eine cast auf ein Objekt, dass nicht dem Zieltypen entspricht, erhält man zur Laufzeit eine ClassCastException
- Damit man dies verhindern kann, ist es möglich, mit instanceof zu prüfen, ob ein Objekt von einer bestimmten Klasse (oder einer davon abgeleiteten) ist

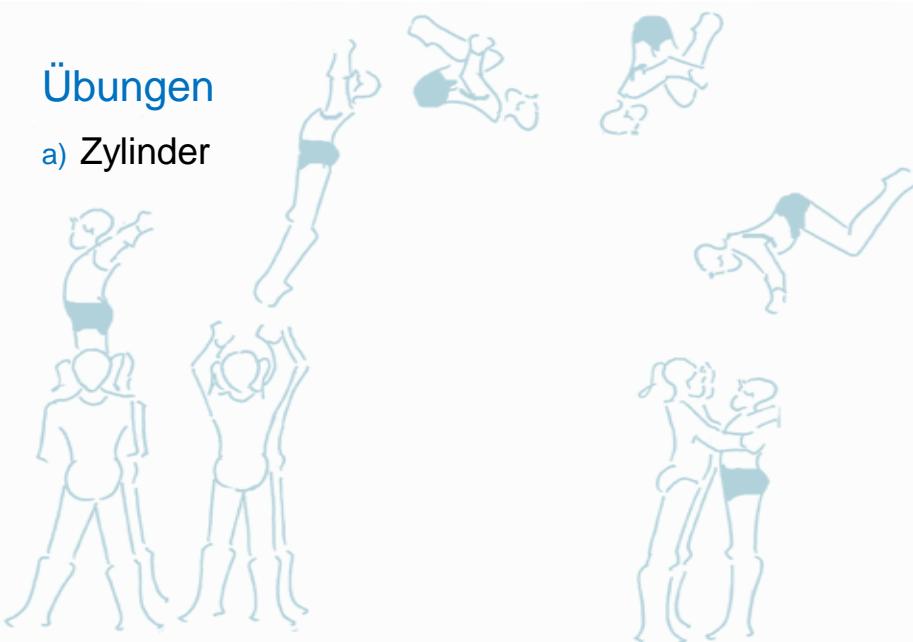
```
public static void main(String[] args) {
    Person person = new Kunde("Peter Müller", 1985, 1234);
    System.out.println(person);

    if (person instanceof Mitarbeiter) {
        Mitarbeiter mitarbeiter = (Mitarbeiter) person;
        mitarbeiter.setLohn(80000);
    } else if (person instanceof Kunde) {
        Kunde kunde = (Kunde) person;
        kunde.setKundenNr(99);
    }
    System.out.println(person);
}
```



## Übungen

### a) Zylinder



## Abschnitt III

### Superklasse Object

179

### Superklasse Object

- ▶ Jede Klasse, die nicht von einer anderen abgeleitet ist, besitzt die **Superklasse Object**
- ▶ Object ist somit direkt oder indirekt die **Basisklasse aller Klassen**
- ▶ Mit der Methode **getClass** kann zum Beispiel der Referenztyp einer Klasse abgefragt werden
- ▶ Mit der Methode **toString()** wird die String Repräsentation des Objektes ausgegeben
  - In der Regel wird diese Methode von den Subklassen überschrieben

180

## clone

- ▶ Mit der Methode **clone** kann ein Kopie eines bestehenden Objektes erzeugt werden
  - Die Attributwerte der Kopie sind dabei exakt die gleichen wie die des ursprünglichen Objekts

```
public class Artikel {

    private int nr;
    private double preis;

    public Artikel(int nr, double preis) {
        this.nr = nr;
        this.preis = preis;
    }

    @Override
    public Artikel clone() {
        return new Artikel(this.nr, this.preis);
    }
    ...
}
```

Mit der optionalen @Override Annotation wird angegeben, dass eine Methode der Oberklasse überschrieben wird.

## equals

- ▶ Mit der Methode **equals(Object obj)** kann geprüft werden, ob ein Objekt die gleichen Inhalte hat, wie das als Parameter übergebene Objekt
- ▶ Wenn equals nicht überschrieben wird, wird für den Vergleich die Objekt Id verwendet
- ▶ Wenn man nur die Attributwerte miteinander vergleichen möchte, so muss die Methode überschrieben werden
- ▶ Mit Eclipse kann die equals Methode (analog den Gettern und Settern) automatisch generiert werden

## equals II

```
public class Artikel {
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Artikel other = (Artikel) obj;
        if (nr != other.nr)
            return false;
        if (Double.doubleToLongBits(preis) != Double.doubleToLongBits(other.preis))
            return false;
        return true;
    }
    ...
}
```

Copyright © iten-engineering.ch

Java Einführung

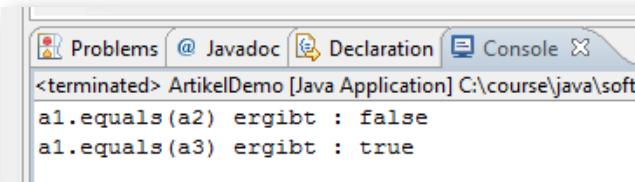
183

183

## Beispiel clone & equals

```
public class ArtikelDemo {
    public static void main(String[] args) {
        Artikel a1 = new Artikel(4711, 12.50);
        Artikel a2 = new Artikel(4712, 12.50);
        Artikel a3 = (Artikel) a1.clone();

        System.out.println("a1.equals(a2) ergibt : " + a1.equals(a2));
        System.out.println("a1.equals(a3) ergibt : " + a1.equals(a3));
    }
}
```



Copyright © iten-engineering.ch

Java Einführung

184

184

## Abschnitt IV

### Finale & Abstrakte Klassen

### Finale Klassen

- ▶ Mit dem Modifier **final** kann verhindert werden, dass von einer Klasse eine andere Klasse abgeleitet werden kann
- ▶ Damit stellt man sicher, dass die vorhandene **Funktionalität nicht** von einer anderen Klasse überschrieben werden kann

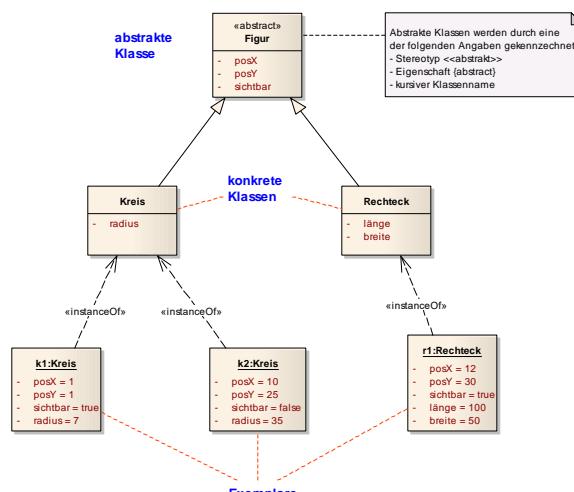
```
// Klasse mit modifier final
public final class Formula {

    // Methode kann nicht mehr überschrieben werden
    public double getSquare(double value) {
        return (value * value);
    }
}
```

## Abstrakte Klassen

- ▶ Klassen von denen **keine konkreten Exemplare** erstellt werden können, d.h. es gibt keine Objekte dieser Klasse, werden als **abstrakte Klassen** bezeichnet
- ▶ Abstrakte Klassen vereinen gemeinsame Eigenschaften und Operationen von Klassen
  - Die Unterklassen erben die gemeinsamen Attribute und Operationen der abstrakten Klassen
- ▶ Im Weiteren besteht die Möglichkeit abstrakte Methoden zu deklarieren
  - In der abstrakten Klasse wird dabei nur die Signatur der Methode deklariert (d.h. Methodename mit Ihren Parametern und Rückgabewert)
  - Die konkreten Unterklassen sind für die Erstellung einer Implementation dieser Methoden zuständig

## Beispiel Abstrakte Klassen



## Beispiel Abstrakte Klasse

```
public abstract class Figure {

    protected int posX;
    protected int posY;
    protected boolean visible;

    public Figure() {
        super();
    }

    public Figure(int posX, int posY, boolean visible) {
        this.posX = posX;
        this.posY = posY;
        this.visible = visible;
    }

    // Abstrakte Methode, muss von Subklassen implementiert werden
    public abstract void move();
}
```

Copyright © iten-engineering.ch

Java Einführung

189

189

## Beispiel Abstrakte Klasse II

```
public class Circle extends Figure {

    protected int radius;

    public Circle() {
        super();
    }

    public Circle(int posX, int posY, boolean visible, int radius) {
        super(posX, posY, visible);
        this.radius = radius;
    }

    @Override
    public void move() {
        System.out.println("move circle");
    }
}
```

Copyright © iten-engineering.ch

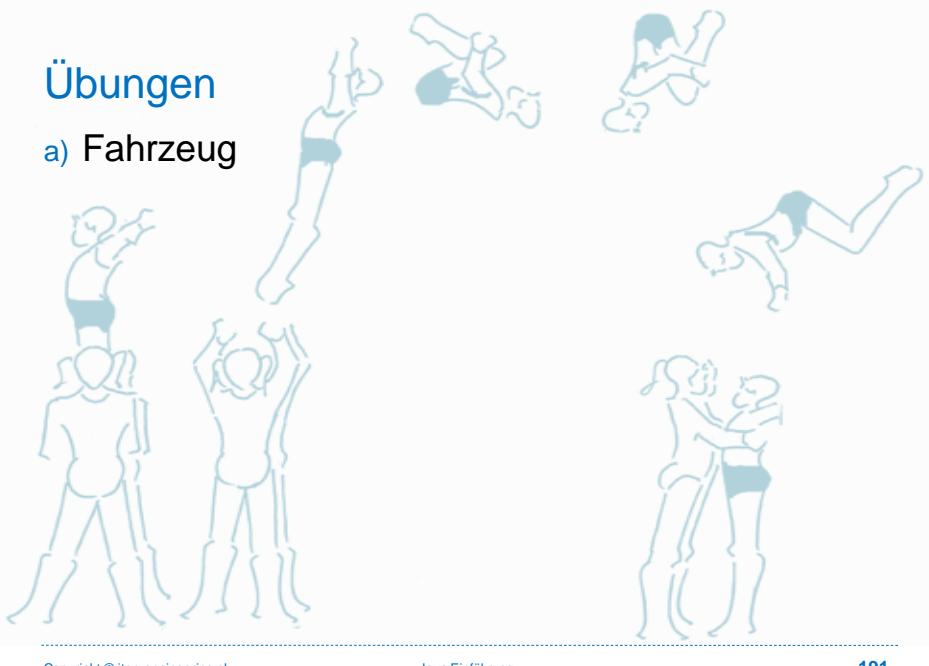
Java Einführung

190

190

## Übungen

### a) Fahrzeug



Copyright © iten-engineering.ch

Java Einführung

191

191

## Kapitel 9

### Packages

Copyright © iten-engineering.ch

Java Einführung

192

192

## Packages

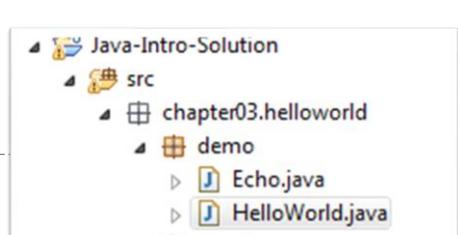
- ▶ Mit Packages werden in Java Klassen **gruppiert**
- ▶ Sammlung von **logisch zusammengehörenden** Klassen, Interfaces und anderen Typen
- ▶ Alle Klassen der Java – API sind in Pakete unterteilt
- ▶ Zum **Beispiel** sind im Package **java.lang** die Math, String und Wrapper-Klassen untergebracht
- ▶ Ein Paketname besteht aus mehreren durch **Punkt getrennten Bezeichnern**

## Packages II

- ▶ Physikalisch werden Pakete auf eine hierarchische Verzeichnisstruktur abgebildet:
  - Jeder Bezeichner entspricht dabei einem Verzeichnis
  - Verzeichnisse sind ineinander verschachtelt

```
package chapter03.demo.helloworld;

public class HelloWorld {
    ...
}
```



## Packages III

- ▶ Der voll qualifizierte Name einer Klasse besteht aus dem Package und Klassennamen
- ▶ Beispiel: `java.lang.String` ist der voll qualifizierte Name der Klasse String
- ▶ Beispiel: `chapter03.demo.helloworld.HelloWorld` ist der voll qualifizierte Name der HelloWorld Anwendung
- ▶ Die Java-API 6.0 zum Beispiel enthält etwa 202 Pakete, die 3777 Klassen und Interfaces umfassen

## Klassenpfad

- ▶ Soll eine Klasse eines andern Packages verwendet werden, so muss sich diese im Klassenpfad befinden
- ▶ Dies wird mit der sogenannten CLASSPATH Variablen angegeben
- ▶ Diese beinhaltet eine Liste mit Pfad und Dateiangabe aller Packages und externen Module (Jar Dateien), die von der Anwendung verwendet werden sollen
- ▶ Innerhalb der Entwicklungsumgebung wird diese automatisch angepasste, wenn ein neues Package dazu kommt

## Import

- ▶ Sobald der Klassenpfad korrekt gesetzt ist, können die Klassen referenziert werden
- ▶ Dabei kann im Programm entweder der **voll qualifizierte Klassenname** verwendet werden
  - Diese Variante kommt in der Regel zum Zug, wenn man gleichnamige Klassen aus unterschiedlichen Packages verwendet
- ▶ Oder man gibt am Anfang vom Programm mit Hilfe eine **Import Statement** an, welche Klassen verwendet werden sollen
  - In der Regel wird diese Variante bevorzugt

## Import II

- ▶ Nur Klassen, die nicht im eigenen Paket liegen müssen importiert werden
- ▶ Ausnahme stellen hier die Bestandteile des Pakets **java.lang** dar
  - Da dieses Paket eines der wichtigsten Pakete der Java-API ist werden alle in ihm enthaltenen Klassen vor jedem Compilerlauf automatisch importiert

```
//importiert alle Klassen und Interfaces
import java.util.*;

//importiert ArrayList
import java.util.ArrayList;
```

## Beispiel Import

```

package demo;

// import von rectangle
import demo.figures.Rectangle;

public class FigureDemo {

    public static void main(String[] args) {

        // Verwendung voll qualifizierter Klassenname (ohne Import)
        chapter08.packages.figures.Circle circle =
            new demo.figures.Circle(3, 7, true, 12);

        // Verwendung unqualifizierter Klassenname (benötigt Import)
        Rectangle rectangle = new Rectangle(15, 25, true, 10, 5);

        circle.move();
        rectangle.move();
    }
}

```

## Statischer Import

- ▶ Gibt es erst seit Java Version 5
- ▶ Innerhalb einer Klasse können ihre statischen Funktionen und Konstanten nur über den Klassennamen angesprochen werden
- ▶ Werden die Bestandteile einer Klasse statisch importiert, so können ihre statischen Methoden und Attribute ohne Klassennamen sofort benutzt werden

## Beispiel Statischer Import

```
// ohne statischen Import
class WithoutStaticImport {

    public static void main(String[] args) {
        System.out.println
            ("Wurzel aus " + Math.PI + " ist " + Math.sqrt(Math.PI));
    }
}

// mit statischem Import
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

class StaticImport {

    public static void main(String[] args) {
        System.out.println
            ("Wurzel aus " + PI + " ist " + sqrt(PI));
    }
}
```

## Zugriffsrechte

- ▶ Wie wir bereits gesehen haben, wird mit den Zugriffsrechten auf Attribute und Methoden die Datenkapselung geregelt
- ▶ Die Modifier **public**, **private** und **protected** bezeichnet man als Access-Modifier
- ▶ Dazu kommt noch die Variante, wenn keine Angaben gemacht werden, dies bezeichnet man als **default**
- ▶ Modifier teilen dem Compiler Informationen mit, wie z.B. Sichtbarkeit, Veränderbarkeit und Lebensdauer von Klassen, Variablen, Konstruktoren, Methoden und Initialisierer

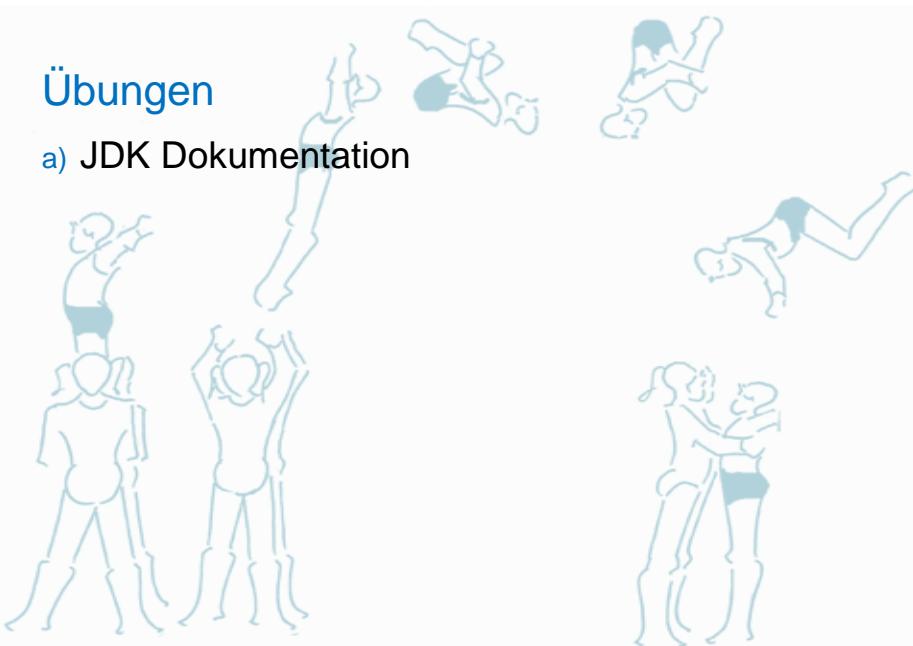
## Access Modifier

- ▶ Access Modifier können bei einer Klasse bei den Variablen, Konstruktoren, Methoden oder der Klasse selbst deklariert werden
- ▶ Wie die folgenden Tabelle zeigt, wird dabei auch die Package Struktur berücksichtigt

| Modifizierer  | Innerhalb der Klasse | Paket- und innere Klassen | Unterklassen | Sonstige Klassen |
|---------------|----------------------|---------------------------|--------------|------------------|
| private       | ja                   | nein                      | nein         | nein             |
| n/a (default) | ja                   | ja                        | nein         | nein             |
| protected     | ja                   | ja                        | ja           | nein             |
| public        | ja                   | ja                        | ja           | ja               |

## Übungen

- a) JDK Dokumentation



## Kapitel 10

### Interfaces und Adapterklassen

---

Copyright © iten-engineering.ch

Java Einführung

205

205

## Abschnitt I

### Einleitung

---

Copyright © iten-engineering.ch

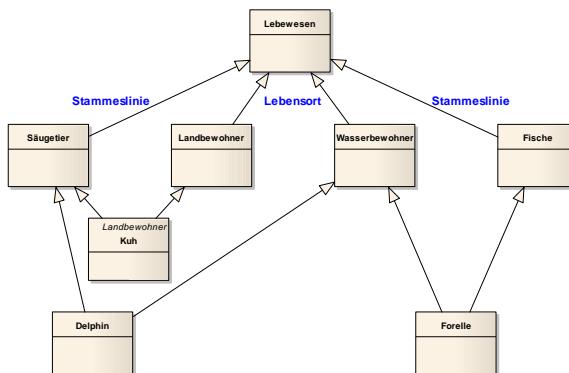
Java Einführung

206

206

## Mehrfachvererbung

- Wenn eine Klasse mehr als eine Oberklasse besitzen kann, so spricht man von Mehrfachvererbung.



## Mehrfachvererbung II

- Bei der Mehrfachvererbung gibt es einige kritische Punkte zu beachten.
- Was passiert zum Beispiel, wenn verschiedene Oberklassen gleichnamige Eigenschaften beinhalten?
  - Von welcher Oberklasse soll die Unterklassse die Eigenschaft nun übernehmen?
  - Dieser Konflikt kann in der Regel nur dadurch vermieden werden, dass die Eigenschaft voll qualifiziert, d. h. inklusive der Bezeichnung für die Oberklasse angesprochen wird.
- Es kann auch die Situation entstehen, dass zwei vererbte Oberklassen ihrerseits eine gemeinsame Oberklasse besitzen.
  - Damit werden Eigenschaften der ersten Oberklasse in zwei Richtungen weitervererbt und später durch Mehrfachvererbung wieder zusammengeführt.

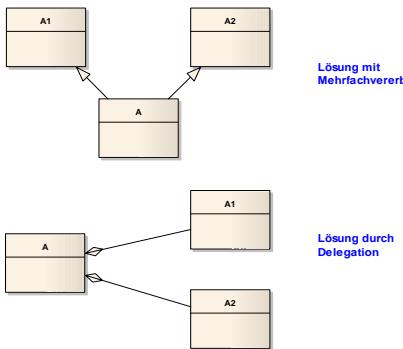
## Mehrfachvererbung III

- ▶ Der Einsatz der Mehrfachvererbung sollte daher nur mit **Vorsicht** und wenn absolut notwendig in Betracht gezogen werden!
- ▶ **Nicht alle Programmiersprachen** unterstützen daher die Mehrfachvererbung
- ▶ In **Java** und **Smalltalk** zum Beispiel ist **keine Mehrfachvererbung** möglich
- ▶ Eine **alternative** zur Mehrfachvererbung bietet sich mit dem Einsatz der **Delegation** oder der Verwendung von **Interface** (Schnittstellen)

## Delegation

- ▶ Vererbung bietet nicht immer nur Vorteile
- ▶ Delegation ist ein alternativer Ansatz, mit dem je nach Problemstellung die Vererbung umgangen werden kann
- ▶ Delegation ist ein Mechanismus, bei dem ein Objekt eine Nachricht nicht vollständig selber interpretiert, sondern an ein anderes Objekt weiterleitet
- ▶ Somit können vorhandene Eigenschaften von anderen Klassen genutzt oder für die Benutzung bereitgestellt werden
- ▶ Durch Delegation erweitert also eine Klasse Ihre Eigenschaften

## Delegation II



- ▶ Die Effekte der Vererbung lassen sich mit dem Beispiel links nachstellen.
- ▶ Hierbei werden Eigenschaften der Oberklassen in separate Klasse ausgegliedert und via Aggregationsbeziehung wieder eingebunden.
- ▶ Die Hauptklasse A delegiert die Nachrichten zu den beiden Beziehungsklassen A1 und A2

## Abschnitt II

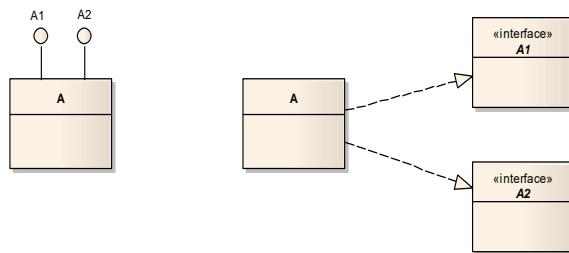
### Interface

## Schnittstellen

- ▶ Die Verwendung von Schnittstellen bei der Programmierung wird auf Englisch als „**Programming against Interfaces**“ bezeichnet
- ▶ Dabei wird mit einer **Schnittstellendefinition** festgelegt, welche Methoden zu einer Schnittstelle gehören
- ▶ Anschliessend wird eine oder mehrere **Klassen** erstellt, welche diese **Schnittstelle implementieren**
- ▶ Schnittstellen repräsentieren also eine Garantie bezüglich der in einer Klasse vorhandenen Methoden
- ▶ Sie geben an, dass alle Objekte, die diese Schnittstelle besitzen, gleich behandelt werden können

## Schnittstellen II

- ▶ Die Schnittstellenbeziehungen sind nicht an den strenge Klassenhierarchien gebunden
- ▶ Die Deklaration erfolgt bei Java mit dem Schlüsselwort **interface**



## Interface

- ▶ Ein Interface besteht aus **statischen Konstanten** und **abstrakten Methoden**, die als public deklariert sind
- ▶ Interfaces dürfen keine Konstruktoren definieren und können daher **nicht instanziert** werden

```
public interface Form {

    /**
     * Flaecheninhalt ermitteln.
     */
    public double getArea();

    /**
     * Umfang ermitteln.
     */
    public double getPerimeter();
}
```

Copyright © iten-engineering.ch

Java Einführung

215

215

## Interface mit Attributen

- ▶ Alle in einem Interface definierten Attribute sind automatisch statische Konstanten, die initialisiert werden müssen

```
public interface UserService {

    /**
     * Global JNDI Name of this service.
     */
    public static final String JNDI_NAME = "ejb/base/UserService";

    public AppUser getAppUser(int userId);

    ...
}
```

Copyright © iten-engineering.ch

Java Einführung

216

216

## Implementation Interface

- ▶ Interfaces werden erstellt, um von Klassen implementiert zu werden
- ▶ Eine Klasse, die ein Interface implementiert, muss alle Interface Methoden überschreiben
- ▶ Dies wird in der Regel mit der @Override Annotation explizit ausgedrückt
- ▶ Wenn eine Klasse nicht alle Methoden eines Interfaces implementiert, muss Sie als abstract deklariert werden

## Beispiel Implementation Interface

```
public class Rectangle implements Form {
    private double width = 1.0;
    private double length = 1.0;

    public Rectangle() {
    }

    // Implementation Interface Methode
    @Override
    public double getArea() {
        return getWidth() * getLength();
    }

    // Implementation Interface Methode
    @Override
    public double getPerimeter() {
        return 2.0 * (getWidth() + getLength());
    }

    ...
}
```

Implementation  
Interface Form

## Interface verwenden

- ▶ Für die Verwendung eines Interface wird eine Variable von Typ des Interface deklariert
- ▶ Dieser Referenz können nun beliebige Objekte von Klassen zugewiesen werden, die dieses Interface implementieren
- ▶ Alle Interface Methoden und Konstanten können so angesprochen werden
- ▶ Die Prüfung eines Typen mit instanceof kann auch für Interface angewendet werden

## Beispiel Interface verwenden

```
public class FormDemo {
    public static void main(String[] args) {
        Form f1 = new Rectangle(2.0, 3.5);
        Form f2 = new Circle(3.0);

        List<Form> forms = new ArrayList<Form>();
        forms.add(f1);
        forms.add(f2);

        for (Form form : forms) {
            System.out.println(form.getClass().getName());
            System.out.println(" area = " + form.getArea());
            System.out.println(" perimeter = " + form.getPerimeter());
        }
    }
}
```

The diagram shows a dashed rectangular box containing the provided Java code. Two blue callout boxes with triangular arrows point from the words "Interface" and "Implementation" to the "Form" class name in the code. To the right of the code, a vertical arrow points upwards towards a screenshot of a Java IDE's console window. The console window displays the output of the program, which includes the names of the implemented classes ("chapter09.interfaces.Rectangle" and "chapter09.interfaces.Circle") and their calculated area and perimeter values.

## Vorteile

- ▶ Die Klassen, die eine Schnittstelle verwenden, kennen die konkrete Implementation nicht
- ▶ Das hat den Vorteil, dass die Implementation ausgetauscht werden kann, ohne dass die Klassen die diese verwenden angepasst werden müssen
- ▶ Man ist also unabhängig von einer konkreten Umsetzung und kann im Nachhinein das Systemverhalten anpassen

## Vorteile II

- ▶ Ein anderer Anwendungszweck ist, wenn eine Klasse mehrere Interface implementiert
- ▶ Die Aufrufer können nun die Klasse über die entsprechenden Schnittstellen aufrufen und verwenden so nur genau die Teile, die sie auch benötigen
- ▶ Mit der Verwendung von Schnittstellen **entkoppelt** man also die **Abhängigkeiten** zwischen Aufrufer und Umsetzung und gewinnt dabei an Flexibilität

## Abschnitt III

### Adapterklassen

223

### Adapterklassen

- ▶ Wenn mehrere Klassen einer **Klassenhierarchie gleiche Interfaces** implementieren, können die zu implementierenden Interface Methoden zum Teil **wiederverwendet** werden
- ▶ Anstatt dass jede Subklasse die Interfaces selber implementiert erstellt man in einem solchen Fall eine **gemeinsame Adapterklasse**
- ▶ Die Adapterklasse implementiert nun alle Interface Methoden
- ▶ Falls nicht alle Methoden implementiert werden sollen, so wird die Adapterklasse als abstrakt deklariert

224

## Adapterklassen II

- ▶ Statt dass die Subklassen nun die Interfaces selber implementieren, erweitern Sie die Adapterklasse
  - Damit können Sie nun die gemeinsamen Implementationen der Adapterklasse nutzen
  - und müssen nur noch die spezifischen Interface Methoden selber implementieren

```
public class FormAdapter implements Form {
    ...
}

public class Circle extends FormAdapter {
    ...
}

public class Rectangle extends FormAdapter {
    ...
}
```

Circle und Rectangle implementieren automatisch auch das Interface Form

## Abschnitt IV

### Interface mit Methodenimplementierung

## Default Methoden

- ▶ Seit **Java 8** ist es möglich in einem Interface eine nicht abstrakte Methode hinzuzufügen
  - Dieses Feature wird auch als **Extension Methods** bezeichnet
- ▶ Hierzu wird die Methodenimplementation mit dem Schlüsselwort **default** gekennzeichnet
- ▶ Damit ist es möglich in bestehenden Systemen **Interfaces** mit neuen Funktionen zu **erweitern**, ohne dass der bestehenden Code bricht!

## Beispiel Default Methoden

```
public interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

- ▶ Innerhalb eines Interface können default Methoden aufeinander zugreifen
- ▶ Im Interface selber kann keine Instanz erstellt werden, dies ist nach wie vor von den Klassen möglich, die das Interface implementieren

## Statische Methoden

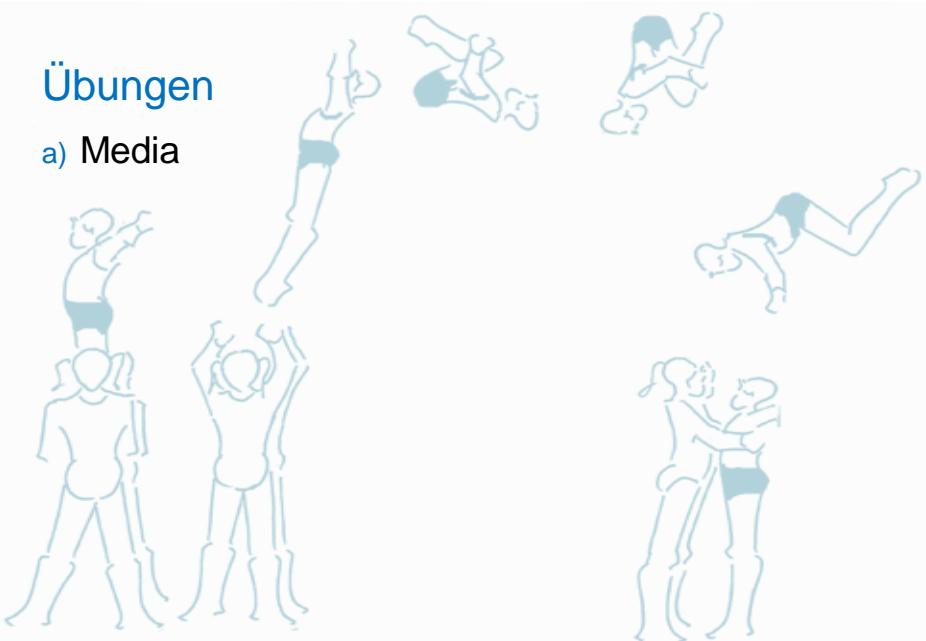
- ▶ Zusammen mit den default Methoden wurden auch **statische Methoden** für Interface eingeführt
- ▶ Hierzu wird die Methodenimplementation mit dem Schlüsselwort **default** gekennzeichnet
- ▶ Die statischen Methoden gehören zum Interface und werden via **Interface Name** aufgerufen
  - Die statischen Methoden dienen u.a. als Hilfsmethoden für default Methoden (Beispiel Comperator)
  - Für reine Utility Klassen mit nur statischen Methoden sind weiterhin normale Klassen zu empfehlen (Beispiel Collectors)

## Vererbung mit Mehrdeutigkeit

- ▶ Mit den neuen Möglichkeiten von Interfaces sind folgenden Scenarien möglich:
  - Eine Klasse implementiert mehrere Interface mit gleichen default Methoden
  - Eine Klasse erweiternt eine Oberklasse und implementiert ein Interface. Die Oberklasse und das Interface beinhalten beide die gleiche Methode
- ▶ Für beide Fälle muss klar geregelt werden, welche Methode wie referenziert und überschrieben werden kann
- ▶ Hierzu gibt es in Java klare **Regeln**

## Übungen

### a) Media



Copyright © iten-engineering.ch

Java Einführung

231

231

## Kapitel 11

### Strings und Wrapper Klassen

Copyright © iten-engineering.ch

Java Einführung

232

232

## Abschnitt I

### Strings

Copyright © iten-engineering.ch

Java Einführung

233

233

### String

- ▶ Für die Abbildung von Zeichenketten bietet Java den Typen **String**
- ▶ String ist ein Referenztyp und besteht aus einer geordneten Folge von Unicode-Zeichen (beginnend mit Index 0)
- ▶ Strings können nicht geändert werden, d.h. Sie sind **immutable** (unveränderbar)
- ▶ Die Klasse String ist **final**, d.h. kann nicht erweitert werden
- ▶ Ein String-Objekt kann entweder mit einem Konstruktor oder als String Literal erstellt werden

```
String s1 = "Hallo";
String s2 = new String ("Welt");
String s3 = s1 + " schöne " + s2;
```

Copyright © iten-engineering.ch

Java Einführung

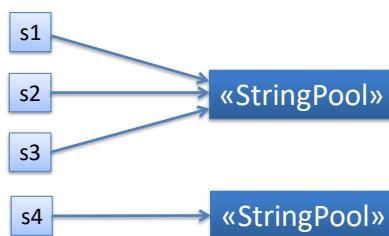
234

234

## String Pool

- ▶ Strings deren Inhalt bereits zum **Zeitpunkt der Kompilierung** feststeht werden nur einmal angelegt und in einem **Pool** verwaltet
- ▶ Existieren mehrere Referenzen auf Strings mit den gleichen Inhalt, so zeigen diese alle auf das gleiche Objekt
- ▶ Strings die **während der Laufzeit** (dynamisch) angelegt werden, wird ein separates **neues Objekt** erstellt

## Beispiel String Pool



```

String s1 = "String Pool";
String s2 = "String Pool";
String s3 = s1;
String s4 = new String("String Pool");

System.out.println(s1 == s2);           // liefert true
System.out.println(s1 == s3);           // liefert true
System.out.println(s1 == s4);           // liefert false
  
```

## Strings vergleichen

- ▶ Für eine **inhaltliche Prüfung** von Strings auf Gleichheit wird (wie bei allen anderen Objekten auch) die **equals** Methode verwendet
- ▶ Für inhaltliche Vergleiche ohne Berücksichtigung der Gross- und Kleinschreibung gibt es zudem die Methode **equalsIgnoreCase**

```
String s1 = "String Compare";
String s2 = "String Compare";
String s3 = s1;
String s4 = new String("String Compare");

System.out.println(s1.equals(s2)); // liefert true
System.out.println(s1.equals(s3)); // liefert true
System.out.println(s1.equals(s4)); // liefert true
```

Copyright © iten-engineering.ch

Java Einführung

237

237

## String Methoden

| Methode                              | Beschreibung                                                                                                                                                       |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| int length()                         | Liefert die Länge der Zeichenkette zurück.                                                                                                                         |
| charAt (int index)                   | Liefert das Zeichen an Position index zurück. Der index muss zwischen 0 und length()-1 liegen, sonst wird eine java.lang.StringIndexOutOfBoundsException geworfen. |
| substring (int begin, int end)       | Liefert den Teilstring zurück, der am Index begin beginnt und am Index end-1 einschliesslich endet.                                                                |
| substring (int begin)                | Liefert den Teilstring von der angegebenen Position bis zum Ende des Strings zurück.                                                                               |
| trim()                               | Liefert den String zurück, der entsteht, wenn alle Leerzeichen am Anfang und am Ende des Strings entfernt werden                                                   |
| toLowerCase()                        | Wandelt den String in Kleinbuchstaben um, liefert ihn zurück                                                                                                       |
| toUpperCase()                        | Methode                                                                                                                                                            |
| replace (char oldChar, char newChar) | Ersetzt alle vorkommen des Zeichens oldChar durch das Zeichen newChar                                                                                              |

Die Klasse String hat noch weitere Methoden. Details hierzu sind in der Javadoc ersichtlich!

Copyright © iten-engineering.ch

Java Einführung

238

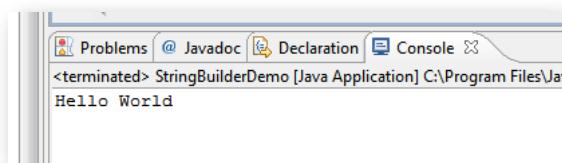
238

## StringBuffer & StringBuilder

- ▶ Mit der Klasse **StringBuffer** können Zeichenketten dynamisch verändert werden
- ▶ Man kann zum Beispiel Texte an einer beliebigen Stelle einfügen oder am Ende anhängen
- ▶ Ab JDK 5 gibt es zusätzlich noch die Klasse **StringBuilder**
  - Diese bietet die gleichen Funktionen wie StringBuffer
  - Die Methoden sind jedoch nicht synchronisiert und daher nicht für konkurrierende Zugriffe gedacht
  - Dafür sind die Zugriffe effizienter (schneller) da der Aufwand für die Synchronisation entfällt

## Beispiel StringBuilder

```
public class StringBuilderDemo {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        sb.append("Hello");
        sb.append(" ");
        sb.append("World");
        System.out.println(sb.toString());
    }
}
```



## Abschnitt II

### Wrapper Klassen

241

### Wrapper Klassen

- ▶ Zu jedem primitiven Datentypen in Java gibt es eine korrespondierende Wrapper-Klasse
- ▶ Diese kapseln die primitiven Werte in einer objektorientierten Hülle
- ▶ Mit new können neue Wrapper Objekte erstellt werden
  - Eine Integer Wrapper Klasse kann zum Beispiel mit einem int Wert oder aber auch mit einem String erstellt werden
- ▶ Wrapper-Klassen verfügen über zahlreiche Methoden und Konstanten

242

## Wrapper Klassen II

| Typ                       | Grösse [Bit]       | Beispiel                      | Default  | Minimum            | Maximum                                 | Wrapper Class |
|---------------------------|--------------------|-------------------------------|----------|--------------------|-----------------------------------------|---------------|
| byte                      | 8                  | byte b = 65;                  | 0        | -128               | 127                                     | Byte          |
| char                      | 16                 | char c = 'A';<br>char c = 65; | '\u0000' | '\u0000'           | '\uffff' <sup>(1)</sup>                 | Character     |
| short                     | 16                 | short s = 65;                 | 0        | -2 <sup>15</sup>   | 2 <sup>15</sup> -1                      | Short         |
| int                       | 32                 | int i = 65;                   | 0        | -2 <sup>31</sup>   | 2 <sup>31</sup> -1                      | Integer       |
| long                      | 64                 | long l = 65L;                 | 0        | -2 <sup>63</sup>   | 2 <sup>63</sup> -1                      | Long          |
| float                     | 32                 | float f = 65f;                | 0.0f     | 2 <sup>-149</sup>  | (2-2 <sup>-23</sup> )·2 <sup>127</sup>  | Float         |
| double                    | 64                 | double d = 65.55;             | 0.0d     | 2 <sup>-1074</sup> | (2-2 <sup>-52</sup> )·2 <sup>1023</sup> | Double        |
| boolean                   | 1                  | boolean b = true;             | false    | n/a                | n/a                                     | Boolean       |
| String<br>(or any Object) | n/a <sup>(2)</sup> | String t = "Hello"            | null     | n/a                | n/a                                     | n/a           |

<sup>(1)</sup> 16 Bit Unicode Zeichenbereich<sup>(2)</sup> Die Grösse für die Reference selber ist ein Implementationsdetail der Java Virtual Machine.

## Beispiele

```
// neues Integer-Objekt mit dem Wert 10 erzeugen
Integer i1 = new Integer(10);

// neues Integer-Objekt mit dem Wert 24 erzeugen
Integer i2 = Integer.valueOf("24");

// gespeicherten int-Wert von i1 liefern
int i = i1.intValue();

// Stringkonvertierung nach int
int j = Integer.parseInt("35");

// Umwandlung einer int-Zahl in einen String
String s = Integer.toString(74);
```

## Autoboxing (boxing/unboxing)

- ▶ Autoboxing gibt es erst seit Java Version 5
- ▶ Vor Version 5 waren die primitiven Datentypen und ihre korrespondierenden Wrapper-Klassen nicht zuweisungskompatibel
- ▶ So war es z. B. nicht möglich eine primitive Integer-Variable einem Integer-Objekt oder umgekehrt zuzuweisen:

```
Integer x = 2;                      // Ungültig

int y = new Integer(3);              // Ungültig
```

## Autoboxing II

- ▶ Unter **Boxing** versteht man die automatische Umwandlung eines primitiven Wertes in ein Wrapper-Klassenobjekt
- ▶ **Unboxing** ist die Umkehrung von Boxing, sprich die automatische Umwandlung eines Wrapper-Klassenobjektes in einen primitiven Wert

```
Boolean b = false;                  // boxing

float f = new Float(3.4f);          // unboxing
```

## Autoboxing III

- ▶ Da die Umwandlung **automatisch** vom Compiler erfolgt, spricht man von Autoboxing
- ▶ Primitive Datentypen und ihre entsprechenden Wrapper-Klassen sind zuweisungskompatibel
- ▶ Vor Version 5 hat musste man die Umwandlung manuelle durchführen

```
// Vor Version 5 manuelle Umwandlung
```

```
Boolean b = new Boolean(false);

Float f1= new Float(3.4f);
float f = f1.floatValue();
```

Copyright © iten-engineering.ch

Java Einführung

247

247

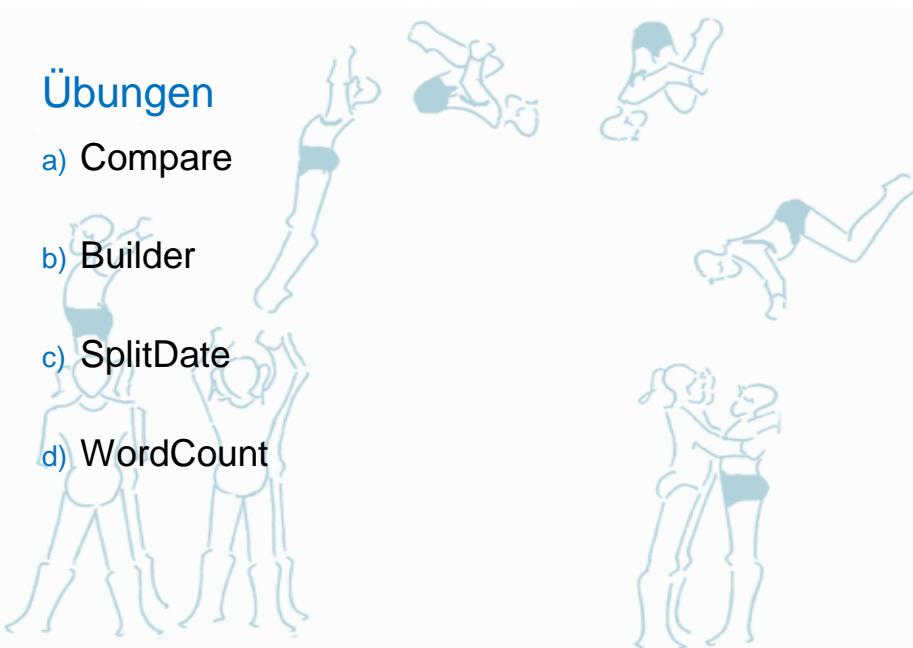
## Übungen

a) Compare

b) Builder

c) SplitDate

d) WordCount



Copyright © iten-engineering.ch

Java Einführung

248

248

## Kapitel 12

### Arrays, Varargs und Enum

---

Copyright © iten-engineering.ch

Java Einführung

249

249

## Abschnitt I

### Arrays

---

Copyright © iten-engineering.ch

Java Einführung

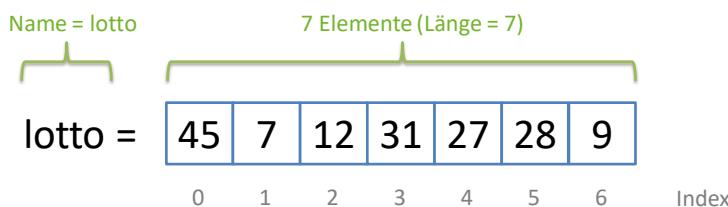
250

250

## Array

- ▶ Ein Array ist eine Sammlung von **gleichen Elementen**
- ▶ Ein Element kann ein **primitiver Datentyp** oder eine **Referenz** sein
- ▶ Ein Array hat **Namen**, einen **Typen** und eine **fixe Länge**
  - Diese Angaben müssen bei der Deklaration des Array gemacht werden
- ▶ Ein einmal erstellter Array kann seine Grösse zur Laufzeit nicht mehr ändern

## Array II



- ▶ Der Array `lotto` hat Platz für 7 Elemente
- ▶ Die Elemente werden via Index referenziert
- ▶ In Java beginnt der Index bei 0

## Definition

- Die Deklaration teilt dem Compiler den **Namen** des Arrays und den **Typ** seiner Elemente
- Mit der Konstruktion wird der benötigte **Speicherplatz** für die Aufnahme der Elemente alloziert
- Beides kann auch in einem Statement erfolgen

```
// Variante A
int[] lottoA;                                // Deklaration
lottoA = new int[7];                           // und Allokation Speicherplatz

// Variante B
int len = 7;                                  // Längenangabe mit Variable
int[] lottoB;                                // Deklaration
lottoB = new int[len];                         // und Allokation Speicherplatz

// Variante C
int[] lottoC = new int[7];                     // Deklaration und Allokation kombiniert
```

Copyright © iten-engineering.ch

Java Einführung

253

253

## Initialisierung

- Immer, wenn ein Array definiert wird, werden seine Felder automatisch mit den Standardwerten ihres Datentyps initialisiert:

| Datentyp                        | Standardwert |
|---------------------------------|--------------|
| Numerische primitive Datentypen | 0            |
| boolean                         | false        |
| Referenztypen                   | null         |

- Will man während der Definition das Array mit anderen Werten initialisieren, so muss die Deklaration, Konstruktion und Initialisierung zu einem einzelnen Schritt kombiniert werden:

```
int[] lottoD = { 45, 7, 12, 31, 27, 28, 9 };
```

Copyright © iten-engineering.ch

Java Einführung

254

254

## Zugriff

- Der Zugriff auf die einzelnen Array Elemente erfolgt mit Hilfe des Index

```

int[] lottoA;                      // Deklaration
lottoA = new int[7];                // und Allokation Speicher

lottoA[0] = 45;                     // Werte zuweisen
lottoA[1] = 7;
lottoA[2] = 12;
lottoA[3] = 31;
lottoA[4] = 27;
lottoA[5] = 28;
lottoA[6] = 9;

System.out.println(lottoA[0]);        // Werte lesen und ausgeben
System.out.println(lottoA[1]);
...
System.out.println(lottoA[6]);

```

Copyright © item-engineering.ch

Java Einführung

255

255

## Zugriff II

- Für die Arbeit mit Arrays eignen sich **for Schleifen** sehr gut
- Hierbei ist zu beachten, dass der Index bei **0** beginnt und bis zur Array **Länge – 1** geht

```

public static void print(int[] lotto) {

    for (int i = 0; i < lotto.length; i++) {
        System.out.print(lotto[i]);
        System.out.print(" ");
    }

    System.out.println();
}

```

Copyright © item-engineering.ch

Java Einführung

256

256

## Fehlerhafte Zugriffe

- ▶ Versucht man eine Zugriff mit einem falschen Index, so erhält man eine **java.lang.ArrayIndexOutOfBoundsException**

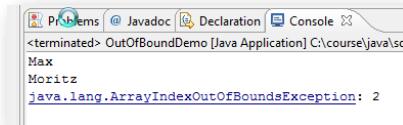
```
String[] name = new String[2]; // Array name mit 2 Elementen

name[0] = "Max";
System.out.println(name[0]);

name[1] = "Moritz";
System.out.println(name[1]);

try {
    name[2] = "Wittwe Bolte";
    System.out.println(name[1]);
}

catch (Exception e) {
    System.out.println(e.toString());
}
```



Copyright © iten-engineering.ch

Java Einführung

257

257

## Beispiel mit Objekten

```
public class Student {
    private String name;
    private String fach;

    public Student(String name, String fach) {
        this.name = name;
        this.fach = fach;
    }

    public String toString() {
        return this.name + ", " + this.fach;
    }
}

Student[] students = new Student[3];

students[0] = new Student("Peter Müller", "Elektrotechnik");
students[1] = new Student("Anita Meyer", "Physik");
students[2] = new Student("Petra Fink", "Informatik");

for (int i = 0; i < students.length; i++) {
    Student student = students[i];
    System.out.println(student.toString());

    // alternative Variante
    System.out.println(students[i].toString());
}
```

Copyright © iten-engineering.ch

Java Einführung

258

258

## Abschnitt II

### Arrays kopieren

### Beispiel int Array kopieren

```
// init
int len = 3;

int[] orig = new int[len];
int[] copy = new int[len];

orig[0] = 7;
orig[1] = 14;
orig[2] = 21;

// copy (primitive)
for (int i = 0; i < orig.length; i++) {
    copy[i] = orig[i];
}

// change orig
orig[1] = 77;

// show result
System.out.println(orig[1]); // Ausgabe ?
System.out.println(copy[1]); // Ausgabe ?
```

Ausgabe ?

## Primitives

- ▶ Bei **primitiven Datentypen** können Arrays mit einer einfache `for` Schleife kopiert werden
- ▶ Alternativ dazu gibt es noch die Methode `System.arraycopy` mit dem ein Array oder Teile davon kopiert werden können
- ▶ Seit **Java 6** existieren in der Klasse `java.util.Arrays` weitere Methoden zum kopieren von Arrays
  - wie zum Beispiel `copyOf`
  - oder `copyOfRange`

## Beispiel Objekt Array kopieren

```
// init
int len = 3;

Mitarbeiter[] orig = new Mitarbeiter[len];
Mitarbeiter[] copy = new Mitarbeiter[len];

orig[0] = new Mitarbeiter(7, "Max");
orig[1] = new Mitarbeiter(14, "Moritz");
orig[2] = new Mitarbeiter(21, "Hans");

// copy (reference)
for (int i = 0; i < orig.length; i++) {
    copy[i] = orig[i];
}

// change orig
orig[1].setNr(77);
orig[1].setName("Fritz");

// show result
System.out.println(orig[1].toString()); // Ausgabe ?
System.out.println(copy[1].toString()); // Ausgabe ?
```

Ausgabe ?

## Reference Types

- ▶ Die einzelnen Elemente im Array sind Referenzen auf die Daten Objekte
- ▶ Wenn nun ein Array Element per Zuweisung einem andern zugewiesen wird, so wird die Referenz kopiert, d.h. beide Elemente zeigen auf das gleiche Objekt
- ▶ Wenn eine Kopie erstellt werden soll, muss ein neues Objekt angelegt werden!

```
orig[1] =
    new Mitarbeiter(14, "Moritz");

copy[1] = orig[1];
```

orig

copy

Mitarbeiter  
nr: 14  
name: Moritz

## Clone

- ▶ Eine Möglichkeit für die Erstellung einer neuen Objekts bietet die `clone` Methode
- ▶ Damit bietet das Objekt selber eine Methode an, um eine Kopie von sich zu erstellen
- ▶ Bei der Default Implementation von `clone` ist zu beachten, dass diese eine `shallow copy` der Daten erstellt
- ▶ Dass heisst, dass bei Klasse mit verschachtelten Elementen wiederum nur die Referenzen kopiert werden

## Clone II

- ▶ Möchte man eine **deep copy** erstellen, so muss man **clone** selber implementieren
- ▶ Der Einsatz von **clone** ist **nicht ganz unproblematisch** und bietet auch Nachteile
- ▶ Daher gibt es auch **alternative Lösungen** für die Erstellung einer deep copy wie zum Beispiel
  - Spezielle Konstruktoren (copy constructor)
  - Factory Methoden
  - Serialisierung und Deserialisierung
- ▶ Weitere Details siehe zum Beispiel:  
[http://en.wikipedia.org/wiki/Clone\\_%28Java\\_method%29#Alternatives](http://en.wikipedia.org/wiki/Clone_%28Java_method%29#Alternatives)

## Beispiel mit clone

```
public class Mitarbeiter {
    ...
    public Mitarbeiter clone() {
        Mitarbeiter result = new Mitarbeiter();
        result.nr = this.nr;
        result.name = new String(this.name);
        return result;
    }
}

// clone
for (int i = 0; i < orig.length; i++) {
    copy[i] = orig[i].clone();
}

// change orig
orig[1].setNr(77);
orig[1].setName("Fritz");

// show result
System.out.println(orig[1].toString()); // Ausgabe ?
System.out.println(copy[1].toString()); // Ausgabe ?
```

## Abschnitt III

### Mehrdimensionale Arrays

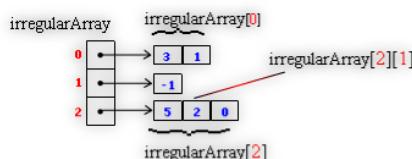
### Mehrdimensionale Arrays

- ▶ Da ein Array Element in Java ein beliebiger Typ sein kann, ist es auch möglich als Element einen Array zu verwenden
- ▶ Damit können zwei- oder mehrdimensionale Arrays definiert werden

```
public class TableDemo {  
    static final int ROWS = 2;  
    static final int COLS = 4;  
  
    public static void main(String[] args) {  
  
        double[][] table = new double[ROWS][COLS];  
  
        for (int row = 0; row < ROWS; row++) {  
            for (int col = 0; col < COLS; col++) {  
                table[row][col] = ...  
            }  
        }  
    }  
}
```

## Irreguläre Arrays

- ▶ Es ist zu beachten, dass die verschachtelten Arrays auch unterschiedliche Längen aufweisen können
- ▶ Man spricht in diesem Zusammenhang von **irregular Arrays**



```

int[][] irregularArray = new int[][] { { 3, 1 }, { -1 }, { 5, 2, 0 } };

int[] elem0 = irregularArray[0];           // array 0 = {3,1}
int[] elem1 = irregularArray[1];           // array 1 = {-1}
int elem1_0 = irregularArray[1][0];        // array 1, element 0 = -1

int[] elem2 = irregularArray[2];           // array 2 = {5,2,0}
  
```

Copyright © iten-engineering.ch

Java Einführung

269

269

## Übungen

a) ArrayValues

b) ArraySearch

c) ArrayCalculation

Copyright © iten-engineering.ch

Java Einführung

270

270

## Abschnitt IV

### Varargs

## Array als Parameter

- ▶ Für die Übergabe der Programm Parameter stellt die main Methoden einen Array von Strings zur Verfügung
- ▶ Die Verwendung von Arrays als Parameter ist aber auch in jeder anderen Methode möglich

```
public static void print(String[] words) {  
  
    for (int i = 0; i < words.length; i++) {  
        System.out.print(words[i]);  
        System.out.print(" ");  
    }  
}
```

## foreach

- ▶ Für die Iteration über Arrays oder Collections gibt es mit **foreach** eine vereinfachte Alternative zur **for** Schleife
  - Dabei werden alle Elemente einmal durchlaufen
  - In jedem Schleifendurchgang wird der aktuelle Wert in einer Variablen zur Verfügung gestellt

```
public static void print(String[] words) {
    for (String word : words) {
        System.out.print(word);
        System.out.print(" ");
    }
}
```

Typ      Variable      Array oder Collection

Copyright © iten-engineering.ch

Java Einführung

273

273

## Beispiel mit Array als Parameter

```
public class ArgsDemo {
    public static void print(String[] words) {
        for (String word : words) {
            System.out.print(word);
            System.out.print(" ");
        }
    }

    public static void main(String[] args) {
        String[] words = { "Heute", "ist", "ein", "schöner", "Tag." };
        print(words);
    }
}
```

- ▶ Die Methode **print** kann mit einer unterschiedlichen Anzahl Wörtern aufgerufen werden
- ▶ Einziger Nachteil ist, dass die Parameter als Array aufbereitet werden müssen

Copyright © iten-engineering.ch

Java Einführung

274

274

## Anonyme Array als Parameter

- ▶ Bei der Übergabe eines Array als Parameter ist es auch möglich, diesen **ad hoc** zu erstellen
- ▶ Man bezeichnet dies als **anonymen Array**

```
public class AnonymDemo {
    public static void print(String[] words) {
        for (String word : words) {
            System.out.print(word);
            System.out.print(" ");
        }
    }

    public static void main(String[] args) {
        print(new String[] { "Heute", "ist", "ein", "schöner", "Tag." });
        print(new String[] { "Finden", "Sie", "nicht", "auch?" });
    }
}
```

Copyright © iten-engineering.ch

Java Einführung

275

275

## Varargs

- ▶ Möchte man auf die Erstellung eines Array als Parameter ganz verzichten, kann man seit Java 5 sogenannte **Varargs** einsetzen
- ▶ Dabei kann man die **gleiche Methode** mit einer **unterschieden Anzahl Parameter** vom gleichen Typ aufrufen

```
public class VarargsDemo {
    public static void print(String... words) {
        for (String word : words) {
            System.out.print(word);
        }
    }

    public static void main(String[] args) {
        print("Heute", "ist", "ein", "schöner", "Tag.");
        print("Finden", "Sie", "nicht", "auch?");
    }
}
```

Varargs

Unterschiedliche Anzahl Parameter

Copyright © iten-engineering.ch

Java Einführung

276

276

## Varargs und Overriding

- ▶ Sofern überladene Methoden zur Verfügung stehen, wird anhand der Signatur entschieden, welche Methode ausgeführt werden soll
- ▶ Gibt es dabei Methoden ohne Varargs, die auf einen Aufruf passen, werden diese bevorzugt

```
public class VarargsDemo {  
    public static void print(String text) {  
        ...  
    }  
  
    public static void print(String... words) {  
        ...  
    }  
  
    public static void main(String[] args) {  
        print("Wort");  
        print("Heute", "ist", "ein", "schöner", "Tag.");  
    }  
}
```

Copyright © iten-engineering.ch

Java Einführung

277

277

## Abschnitt V

### Enums

Copyright © iten-engineering.ch

Java Einführung

278

278

## Enum

- ▶ Seit Java 5 können mit dem Schlüsselwort `enum` beliebige Aufzählungen definiert werden
- ▶ Die Definition erfolgt dabei analog einer Klasse, aber mit dem Schlüsselwort `enum`

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

Day today = Day.MONDAY;
Day tomorrow = Day.TUESDAY;

System.out.println(today);                                // Ausgabe MONDAY
System.out.println(today.name());                         // Ausgabe MONDAY
System.out.println(today.ordinal());                     // Ausgabe: 1

System.out.println(tomorrow);                            // Ausgabe THUESDAY
System.out.println(tomorrow.ordinal());                  // Ausgabe: 2
```

## Enum II

- ▶ Ein praktische Methode ist die statische Methode `values()`, welche einen Array von allen Enum Werten zurückliefert
- ▶ Enum Werte können auch direkt in Switch Statements verwendet werden
- ▶ Dadurch wird die Auswertung von Enum's sehr einfach

## Beispiel mit for Loop und switch

```
// Iteration über alle Werte von Day
for (Day day : Day.values()) {

    // Auswertung mit switch
    switch (day) {

        case MONDAY:
            System.out.println("Eine neue Woche beginnt");
            break;

        case TUESDAY:
            System.out.println("Die Woche dauert noch so lange");
            break;

        ...

        case FRIDAY:
            System.out.println("Am Freitag gehen wir ins Kino");
            break;

        case SATURDAY:
        case SUNDAY:
            System.out.println("Yes, weekend!");
            break;
    }
}
```

## Definition

- ▶ Jedes Enum Objekt besitzt automatisch einige Standardfunktionen, die von der Oberklasse `java.lang.Enum` geerbt werden
- ▶ Der Unterschied zu einer Klassen Definition ist, dass bei einem Enum **keine Unterklassen und Oberklassen** möglich sind
- ▶ Sonst erlaubt enum auch die Definition von **Methoden** und **Variablen**
- ▶ Ein Enum kann zum Beispiel auch **Interface's implementieren**

## Methoden Auswahl

- ▶ **final int ordinal()**
  - Liefert die zur Konstante gehörige ID. Im Allgemeinen ist diese Ordinalzahl nicht wichtig, aber besondere Datenstrukturen wie EnumSet oder EnumMap nutzen diese eindeutige ID. Die Reihenfolge der Zahlen ist durch die Reihenfolge der Angabe gegeben.
- ▶ **final String name()**
  - Liefert den Namen der Konstanten. Da die Methode – wie viele andere der Klasse – final ist, lässt sich der Name nicht ändern.
- ▶ **String toString()**
  - Liefert den Namen der Konstanten. Die Methode ruft standardisiert name() auf, weil sie aber nicht final ist, kann sie überschrieben werden.

## Beispiel Definiton

```
public enum Creature {
    // Definition Werte mit Aufruf des Konstruktors
    SNAKE(0), LIZARD(4), FISH(0), SPIDER(8), INSECT(6);

    // Attribut
    private int legs;

    // Konstruktor
    Creature(int legs) {
        this.legs = legs;
    }

    // Methoden
    public int getLegs() {
        return legs;
    }

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

## Beispiel Definition II

```
package chapter11.enums;

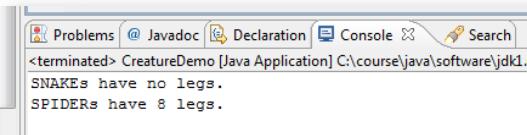
import static chapter11.enums.Creature.SNAKE;
import static chapter11.enums.Creature.SPIDER;

public class CreatureDemo {

    public static void main(String[] args) {

        Creature oneCreature = SNAKE;
        Creature anotherCreature = SPIDER;

        System.out.println(oneCreature.getDescription());
        System.out.println(anotherCreature.getDescription());
    }
}
```



Copyright © iten-engineering.ch

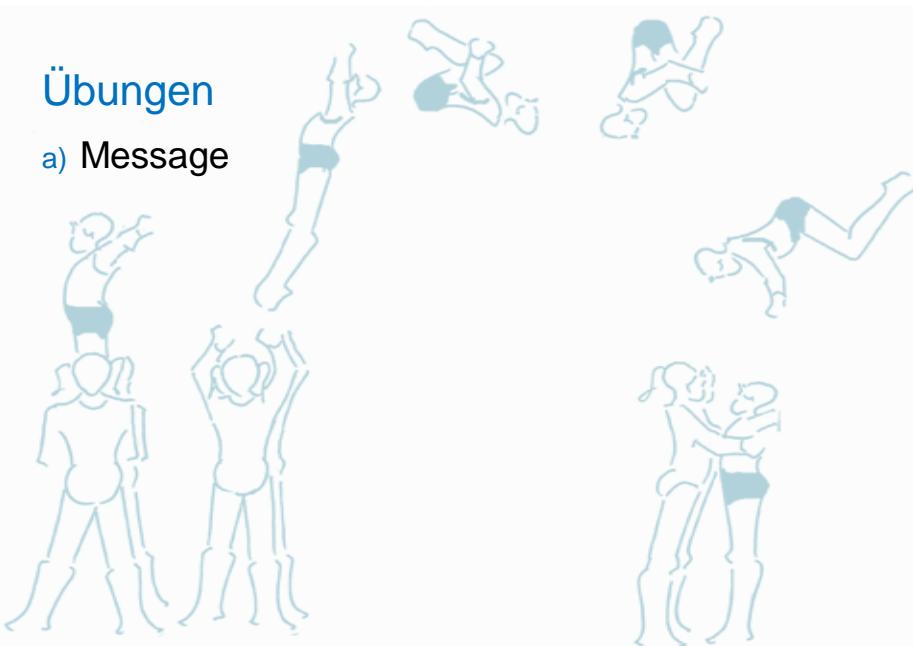
Java Einführung

285

285

## Übungen

### a) Message



Copyright © iten-engineering.ch

Java Einführung

286

286

## Kapitel 13

### Collection Framework

---

Copyright © iten-engineering.ch

Java Einführung

287

287

## Abschnitt I

### Übersicht

---

Copyright © iten-engineering.ch

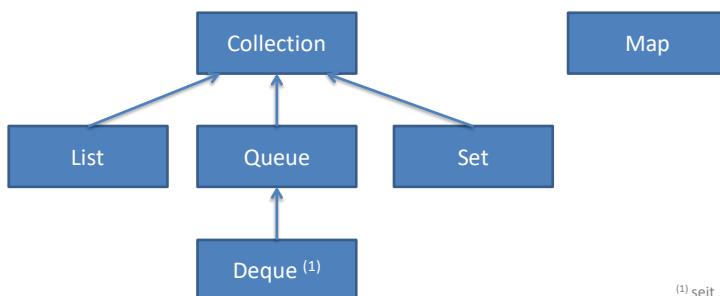
Java Einführung

288

288

## Collections

- ▶ Collections in Java sind Datenstrukturen,
  - die als Container aufgefasst werden können,
  - in denen Objekte gespeichert, verwaltet und manipuliert werden können
- ▶ Den Kern des Collections-Frameworks bilden folgende Interfaces:



(1) seit Java 6

## Collections II

| Interface  | Beschreibung                                                                                                                                                                              |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Collection | Basis Interface mit gemeinsamen Methoden. Ausser Map implementieren alle Collections dieses Interface.                                                                                    |
| List       | Beliebig grosse Liste mit Elementen unterschiedlicher Typen. Kann gleiche Elemente enthalten. Zugriff sequentiell oder wahlweise.                                                         |
| Set        | Eine Menge von Elementen ohne Duplikate. Zugriff via Mengenoperationen.                                                                                                                   |
| Queue      | Spezielle Liste auf die nur sequentiell zugegriffen werden kann. Wird häufig als Warteschlange / FIFO (first in, first out) eingesetzt. Zugriff auf beliebige Elemente ist nicht erlaubt. |
| Deque      | Double ended queue, spezielle Queue bei der Elemente am Anfang oder Ende hinzugefügt oder entfernt werden können. Zugriff auf beliebige Elemente ist auch hier nicht erlaubt.             |
| Map        | Menge von zusammengehöriger Objektpaare, die jeweils als Schlüssel / Wert Paare verwaltet werden                                                                                          |

## Collections III

- ▶ Die Collections bieten eine grosse Vielfalt von Anwendungsmöglichkeiten für die Verarbeitung von Daten
- ▶ Zu beachten ist, dass nicht alle Collection Implementierungen synchronisiert sind, d.h. parallele Zugriffe werden nicht automatisch kontrolliert
- ▶ Collections können beliebige Datentypen aufnehmen, möchte man diesen beschränken, so kann dies mit Hilfe der Generics (seit Java 5) explizit angegeben werden

```
// Liste mit beliebigen Objekt Typen
List mixed = new ArrayList();

// Liste mit Elementen vom Typ Integer
List<Integer> zahlen = new ArrayList<Integer>();
```

## Collection Interface (Javadoc)

### Method Summary

boolean [add\(E e\)](#)

Ensures that this collection contains the specified element (optional operation).

boolean [addAll\(Collection<? extends E> c\)](#)

Adds all of the elements in the specified collection to this collection (optional operation).

void [clear\(\)](#)

Removes all of the elements from this collection (optional operation).

boolean [contains\(Object o\)](#)

Returns true if this collection contains the specified element.

boolean [containsAll\(Collection<?> c\)](#)

Returns true if this collection contains all of the elements in the specified collection.

## Collection Interface II

### Method Summary

|                                   |                                                 |                                                                                                                     |
|-----------------------------------|-------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| boolean                           | <a href="#">equals(Object o)</a>                | Compares the specified object with this collection for equality.                                                    |
| int                               | <a href="#">hashCode()</a>                      | Returns the hash code value for this collection.                                                                    |
| boolean                           | <a href="#">isEmpty()</a>                       | Returns true if this collection contains no elements.                                                               |
| <a href="#">Iterator&lt;E&gt;</a> | <a href="#">iterator()</a>                      | Returns an iterator over the elements in this collection.                                                           |
| boolean                           | <a href="#">remove(Object o)</a>                | Removes a single instance of the specified element from this collection, if it is present (optional operation).     |
| boolean                           | <a href="#">removeAll(Collection&lt;?&gt;c)</a> | Removes all of this collection's elements that are also contained in the specified collection (optional operation). |

## Collection Interface III

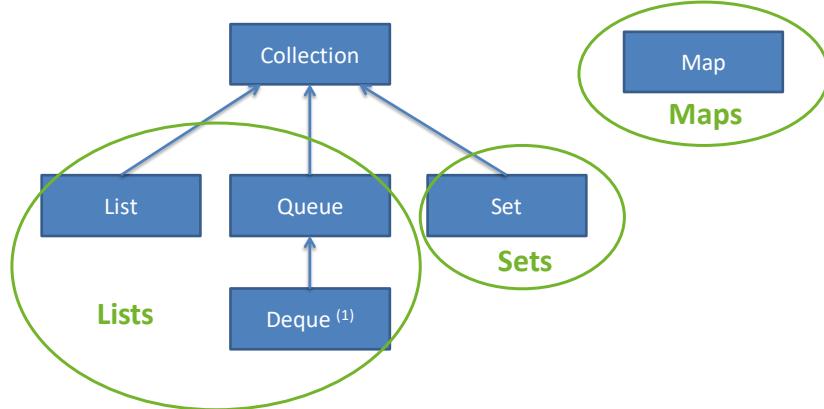
### Method Summary

|                               |                                                 |                                                                                                                                            |
|-------------------------------|-------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| boolean                       | <a href="#">retainAll(Collection&lt;?&gt;c)</a> | Retains only the elements in this collection that are contained in the specified collection (optional operation).                          |
| int                           | <a href="#">size()</a>                          | Returns the number of elements in this collection.                                                                                         |
| <a href="#">Object[]</a>      | <a href="#">toArray()</a>                       | Returns an array containing all of the elements in this collection.                                                                        |
| <a href"="">&lt;T&gt; T[]</a> | <a href="#">toArray(T[] a)</a>                  | Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. |

- ▶ Die in der Javadoc mit **optional operation** gekennzeichneten Methoden müssen nicht von allen Klassen implementiert werden
- ▶ Bei diesen Klassen wird beim Aufruf solcher Methoden eine Exception ausgelöst

## Kategorien

- ▶ Die Collections werden in folgende Kategorien unterteilt:



## Kategorien II

- ▶ Lists
  - sind Sammlungen, die dynamischen Arrays ähneln
  - die Elemente sind geordnet und indiziert
  - Duplikate sind erlaubt
- ▶ Sets
  - repräsentieren (mathematische) Mengen, die keine Duplikate zulassen
  - Es gibt sortierte und unsortierte (geordnet und nicht geordnete) Sets
- ▶ Maps
  - Sortierte oder unsortierte Sammlung von Paaren
  - Jedes Paar besteht aus einem eindeutigen Schlüssel und einem zugehörigen Objekt (Wert)
  - Sind vergleichbar mit assoziativen dynamischen Arrays
  - Werden mithilfe von Hash-Strukturen implementiert

## Abschnitt II

### Lists

297

### List

- ▶ Listen sind vergleichbar mit dynamisch wachsenden Arrays
- ▶ Sind geordnet (sie speichern ihre Elemente in der Reihenfolge, in der sie eingefügt wurden)
- ▶ Jedes Element hat einen Index
- ▶ Die Indizierung fängt, wie bei Arrays, bei 0 an
- ▶ Indexbasierter Zugriff auf Listenelemente ist möglich
- ▶ Das Einfügen und Entfernen von Elementen an beliebiger Stelle in Listen ist möglich
- ▶ Listen können **null** Objekte und Duplikate speichern
- ▶ Die Länge einer Liste ist theoretisch unbeschränkt

298

## List II

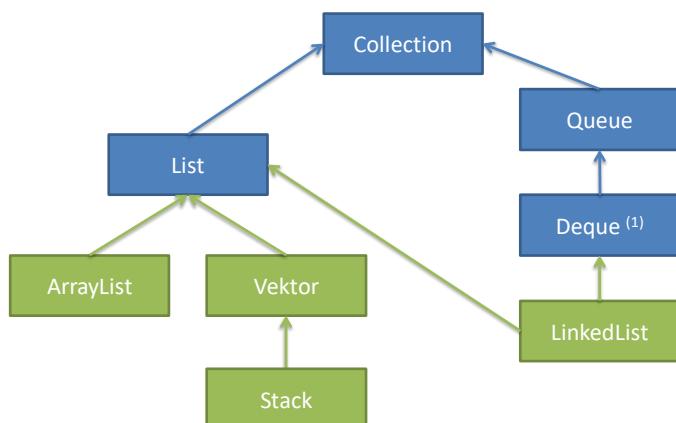
- Das List Interface definiert zusätzliche Methoden, so zum Beispiel die folgenden, welche für den wahlfreien Zugriff via Index:

### Method Summary

|                                             |                                                                                                                                   |
|---------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <code>void add(int index, E element)</code> | Inserts the specified element at the specified position in this list (optional operation).                                        |
| <code>E get(int index)</code>               | Returns the element at the specified position in this list.                                                                       |
| <code>int indexOf(Object o)</code>          | Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| <code>E set(int index, E element)</code>    | Replaces the element at the specified position in this list with the specified element (optional operation).                      |

## Implementationen

- Anbei einige Beispiele von **Implementationen**:



## Beispiel ArrayList

```
// Deklaration Liste mit Elementen vom Typ Integer
List<Integer> numbers = new ArrayList<Integer>();

// Elemente hinzufügen
numbers.add(2); // index 0
numbers.add(4); // index 1
numbers.add(6); // index 2, oops, wrong entry
numbers.add(8); // index 3

// Elemente lesen
Integer wrongEntry = numbers.get(2); // wrongEntry = 6

// Element löschen
numbers.remove(2);

// Elemente auslesen via foreach Schleife
for (Integer number : numbers) {
    System.out.println(number);
}
```

Copyright © iten-engineering.ch

Java Einführung

301

301

## ArrayList vs. Vector

- ▶ Implementieren beide das List-Interface
- ▶ Sie unterscheiden sich jedoch darin, dass die Methoden von Vector synchronisiert sind
- ▶ Mehrere Threads auf einen Vector gleichzeitig zugreifen können ohne die Integrität der Daten zu verletzen
- ▶ Aufgrund der Synchronisation sind Vektoren langsamer als ArrayLists und sollten daher nur verwendet werden wenn mehrere Threads auf den Vektor konkurrierend zugreifen sollen
- ▶ Vectors sind keine Vektoren im mathematischen Sinne sondern Listen

Copyright © iten-engineering.ch

Java Einführung

302

302

## Beispiel Vector

```
public class VectorDemo {

    public static void main(String args[]) {

        List<String> messages = new Vector<String>();

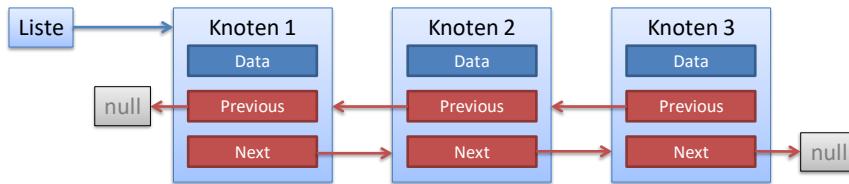
        messages.add("Kuchen");
        messages.add(null); → null Referenz
        messages.add("Pizza");
        messages.add("Kuchen"); → Duplikat
        messages.add("Salat");

        for (String message : messages) {
            System.out.println(message);
        }
    }
}
```

## ArrayList vs. LinkedList

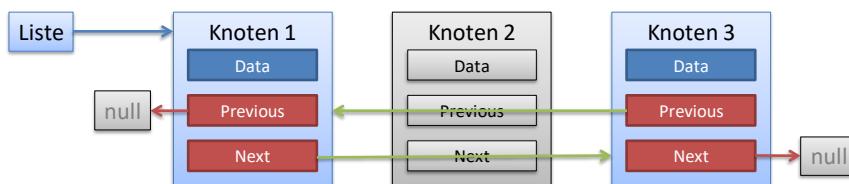
- ▶ Eine **ArrayList** verwaltet Ihre Daten intern in einem Array
  - Das bedeutet, dass beim Einfügen oder Löschen von Elementen, die Positionen **aller anderen Konten** u.U. wechseln und angepasst werden müssen
- ▶ Eine **LinkedList** hingegen verwaltet Ihre Daten in einer **doppelt verketteten Liste**
  - Beim Einfügen oder Löschen von Elementen sind so nicht alle Elemente betroffen, sondern nur die **Knoten vor** und **nach** dem betroffenen Element
  - Bei grossen Datenmengen mit häufigen Einfüge- und Löschoperationen sind daher **LinkedList** performanter

## Linked List



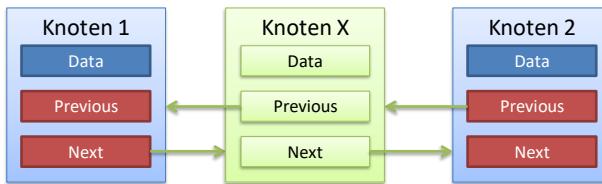
- ▶ Jeder Konten enthält drei Referenzen
  - eine Referenz auf das Objekt mit den Daten
  - Eine Referenz auf den vorherigen Knoten
  - Eine Referenz auf den nächsten Knoten
- ▶ Die Konten am Anfang und Ende enthalten jeweils nur eine gültige Referenz, die andere ist null.

## Linked List – Element löschen



- ▶ Wird jetzt zum Beispiel Knoten 2 gelöscht, so müssen einzig die **Referenzen** angepasst werden
  - Next von Knoten 1 zeigt neu auf Knoten 3
  - Previous von Konten 3 zeigt neu auf Knoten 1

## Linked List – Element einfügen



- ▶ Wird ein neuer Knoten eingefügt, so müssen wiederum die **Referenzen** angepasst werden
  - Next von Knoten 1 zeigt neu auf Knoten X
  - Previous von Konten 2 zeigt neu auf Knoten X
  - Next und Previous von Konten X werden entsprechend gesetzt

## Iterator

- ▶ Collections unterstützen auch das Iterator Interface mit dem Daten sequentiell durchlaufen werden können
- ▶ Dabei kann man sich einen Zeiger vorstellen, der auf ein Element in der Collection zeigt
- ▶ Ausgehend von dieser aktuellen Position können die folgenden Methoden ausgeführt werden:

| Method Summary |                                                                                                                                    |
|----------------|------------------------------------------------------------------------------------------------------------------------------------|
| boolean        | <a href="#">hasNext()</a><br>Returns true if the iteration has more elements.                                                      |
| E              | <a href="#">next()</a><br>Returns the next element in the iteration.                                                               |
| void           | <a href="#">remove()</a><br>Removes from the underlying collection the last element returned by the iterator (optional operation). |

## Beispiel Iterator

```
// Deklaration Liste mit Elementen vom Typ Integer
List<Integer> numbers = new ArrayList<Integer>();

// Elemente hinzufügen
numbers.add(2);
numbers.add(4);
numbers.add(8);

// Verwendung Iterator mit while Schlaufe
Iterator<Integer> it = numbers.iterator();

while (it.hasNext()) {
    Integer number = (Integer) it.next();

    System.out.println(number);
}
```

Initialisierung

Abfrage

Nächstes Element lesen

Copyright © iten-engineering.ch

Java Einführung

309

309

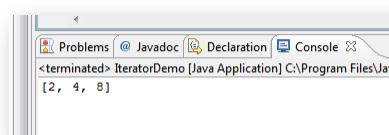
## Ausgabe mit `toString()`

- ▶ Kleine Listen können für Testzwecke auch mit der `toString()` Methode ausgegeben werden
- ▶ Dabei werden die Daten in **[] Klammern** ausgegeben
- ▶ Die Ausgabe der Daten erfolgt dabei über `toString()` Methode des jeweiligen Objektes

```
// Deklaration Liste mit Elementen vom Typ Integer
List<Integer> numbers = new ArrayList<Integer>();

// Elemente hinzufügen
numbers.add(2);
numbers.add(4);
numbers.add(8);

// Ausgabe mit der toString() Methode
System.out.println(numbers.toString());
```



Copyright © iten-engineering.ch

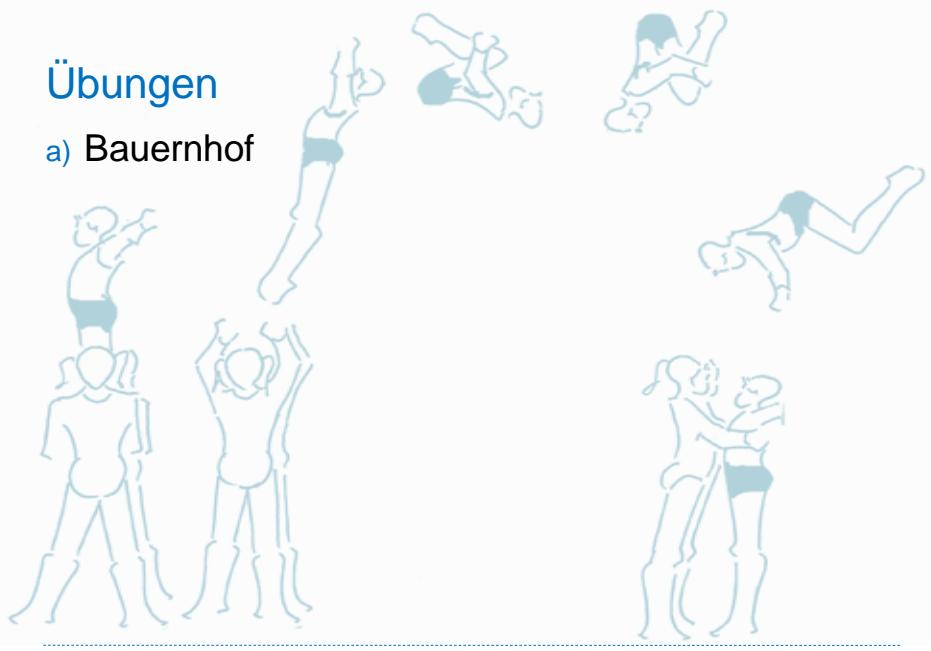
Java Einführung

310

310

## Übungen

### a) Bauernhof



Copyright © iten-engineering.ch

Java Einführung

311

311

## Abschnitt III

### Sets

Copyright © iten-engineering.ch

Java Einführung

312

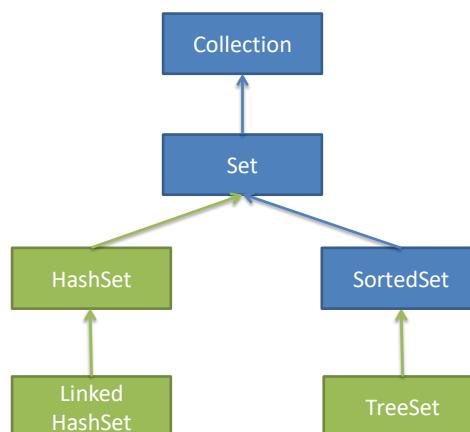
312

## Set

- ▶ Ein Set ist im **mathematischen** Sinn eine **Menge**
- ▶ Ein Menge ist eine Sammlung von Elementen, die **keine Duplikate** zulässt
- ▶ Bevor ein Element in eine Menge eingefügt werden kann, wird zunächst **überprüft**, ob es bereits darin enthalten ist
- ▶ Im Gegensatz zu Listen ist bei Mengen ein **indexbasierter Zugriff** auf die Elemente **nicht möglich**
- ▶ Es gibt **sortierte** und **unsortierte** Mengen

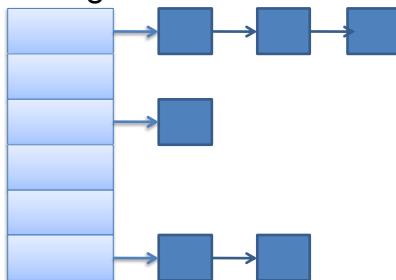
## Implementationen

- ▶ Anbei einige Beispiele von **Implementationen**:



## Hash Tabellen

- ▶ Hash Tabellen sind für schnelle Zugriffe konzipiert
- ▶ Die Position an der ein Element abgelegt wird, hängt von seinem Hashcode ab
- ▶ Die Berechnung des Hashcode ist schnell und hängt einzig vom Zustand des jeweiligen Elements ab
- ▶ Eine Hash Tabelle ist ein Array von verketteten Listen
- ▶ In eine verkettete Liste werden nur Elemente mit identischem Hashcode abgelegt



## HashSet

- ▶ Implementation einer Hashtabelle, die aber keine Duplikate zulässt
- ▶ Lässt das Einfügen von null Objekten zu
- ▶ Die Elemente eines HashSet sind **nicht geordnet**
- ▶ Ein HashSet verlässt sich auf die Implementierung der `hashCode()` Methode ihrer Objekte
- ▶ Eine `LinkedHashSet` hat die gleichen Eigenschaften wie ein HashSet ist aber **geordnet**

## Beispiel HashSet

Klasse HashSet

Interface Set

Typ Person

```

Set<Person> persons = new HashSet<Person>();

Person astrid = new Person(25, "Astrid", "Lindgren");
Person pipi = new Person(25, "Pipi", "Langstrumpf");
Person annika = new Person(25, "Annika", "Halström");

persons.add(astrid);
persons.add(pipi);
persons.add(annika);

System.out.println("Personen:");
for (Person person : persons) {
    System.out.println(person.toString());
}
System.out.println();

System.out.print("Enthält die Liste Pipi: ");
System.out.println(persons.contains(pipi));

System.out.print("Pipi noch einmal eingefügt: ");
System.out.println(persons.add(pipi));

```

true

false

Copyright © iten-engineering.ch Java Einführung 317

317

## Beispiel HashSet II

- ▶ Mit der Methode `hashCode` wird der Hashcode eines Elements berechnet
- ▶ Diese Methode wird von der Klasse Object zur Verfügung gestellt
- ▶ Verwendet man **eigene Objekte** die in einem Set verwaltet werden sollen, so ist die Methode zu überschreiben
- ▶ Gleichzeitig überschreibt man auch die Methode `equals`, damit die Objekte inhaltlich verglichen werden können
- ▶ In Eclipse kann man sich beide Methoden, **automatisch generieren lassen**
  - Dabei kann man angeben, welche Felder für die Berechnungen mit einbezogen werden sollen

Copyright © iten-engineering.ch Java Einführung 318

318

## Beispiel HashSet III

```
public class Person {

    private int nr;
    private String firstname;
    private String lastname;

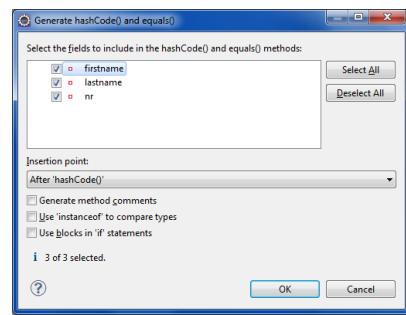
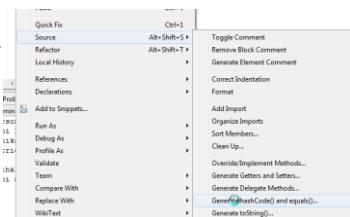
    // Konstruktoren

    @Override
    public int hashCode() {
        ...
    }

    @Override
    public boolean equals(Object obj) {
        ...
    }

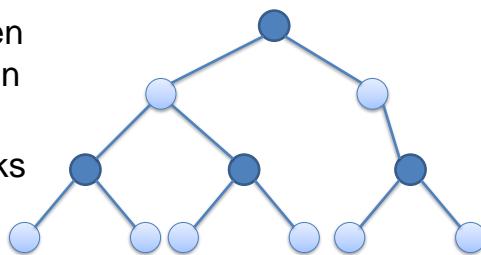
    // weitere Methoden

    // Getter & Setter
}
```



## Baumstrukturen

- ▶ Eine Alternative zur Hash Tabellen bieten Baumstrukturen (Trees)
- ▶ Ausgehend von einem Wurzelknoten hat man verschiedene Unterknoten die ihrerseits wiederum Unterknoten haben
- ▶ Die Abbildung deutet einen binären Tree an:
- ▶ Dabei hat jeder Knoten max. zwei Unterknoten
- ▶ Die Elemente werden jetzt, je nach Wert links oder rechts von den Knoten abgelegt



## TreeSet

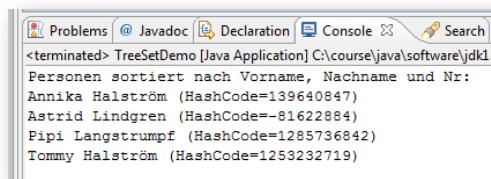
- ▶ Speichert die Elemente **geordnet** in einer **Baumstruktur**
- ▶ Die Elemente müssen daher eine **Sortierreihenfolge** festlegen
- ▶ Neue Daten können hinzugefügt werden, ohne dass das TreeSet neu sortiert werden muss
- ▶ Lässt weder Duplikate noch null Objekte zu
- ▶ Auch wenn **Zugriffe** über einen Baum langsamer sind als bei einer Hash Tabelle, so sind sie doch wesentlich **schneller** als bei **Arrays** oder verketteten **Listen**

## Beispiel TreeSet

```
public static void main(String[] args) {
    Set<ComparablePerson> persons = new TreeSet<ComparablePerson>();

    persons.add(new ComparablePerson(25, "Astrid", "Lindgren"));
    persons.add(new ComparablePerson(42, "Pipi", "Langstrumpf"));
    persons.add(new ComparablePerson(7, "Tommy", "Halström"));
    persons.add(new ComparablePerson(7, "Annika", "Halström"));

    System.out.println("Personen sortiert nach Vorname, Nachname und Nr:");
    for (ComparablePerson person : persons) {
        System.out.println(person.toString());
    }
}
```



## Comparable

- ▶ Für das Festlegen der Sortierreihenfolge müssen die Objekte welche in einem TreeSet verwaltet werden, das Comparable Interface implementieren
  - Bei den vordefinierten Klassen wie z.B. String ist das Interface bereits implementiert
  - Bei eigenen Klassen, die in einem TreeSet verwaltet werden sollen, muss man das Interface selber implementieren
- ▶ Das Interface definiert einzig die Methode compareTo für den Vergleich des aktuellen Objektes mit einen anderen
  - Die Rückgabe ist 0 wenn die Objekte gleich sind, < 0 wenn das aktuelle Objekt kleiner ist und > 0 im anderen Fall

### Method Summary

int [compareTo\(T o\)](#)

C.compares this object with the specified object for order.

## Beispiel compareTo

```
public class ComparablePerson extends Person
  implements Comparable<ComparablePerson> {

  @Override
  public int compareTo(ComparablePerson obj) {
    final int LESS = -1;
    final int EQUAL = 0;
    final int GREATER = 1;

    // Wenn obj null ist oder nicht vom gleichen Typ,
    // so ist keine inhaltliche Prüfung möglich
    if (obj == null || this.getClass() != obj.getClass()) {
      return LESS;
    }

    // Wenn obj auf das gleiche Objekt zeigt,
    // dann ist die inhaltliche Prüfung nicht notwendig
    if (this == obj) {
      return EQUAL;
    }

    ...
  }
}
```

Teil 1

Eigene Klasse  
implementiert  
das Interface  
Comparable

## Beispiel compareTo II

```
// Die inhaltliche Prüfung soll in folgender Reihenfolge
// gemacht werden: firstname, lastname, nr
int result;

result = this.firstname.compareTo(obj.firstname);
if (result != EQUAL)
    return result;

result = this.lastname.compareTo(obj.lastname);
if (result != EQUAL)
    return result;

if (this.nr < obj.nr)
    return LESS;

if (this.nr > obj.nr)
    return GREATER;

// Alle Felder sind gleich.
return EQUAL;
}
```

Teil 2

firstname & lastname sind vom Typ String. Die Klasse String stellt bereits eine compareTo Methode zur Verfügung, diese kann wiederverwendet werden.

Das Feld nr ist vom Typ int. Hier können die Werte direkt verglichen werden.

## Weitere Methoden

- ▶ TreeSet hat viele weitere nützliche Funktionen
- ▶ Zum Beispiel auch solche, die eine Sicht auf eine Teilmenge des Tree zur Verfügung stellen:

### Method Summary

|                                    |                                                    |                                                                                                                      |
|------------------------------------|----------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <a href="#">E</a>                  | <a href="#">first()</a>                            | Returns the first (lowest) element currently in this set.                                                            |
| <a href="#">E</a>                  | <a href="#">last()</a>                             | Returns the last (highest) element currently in this set.                                                            |
| <a href="#">SortedSet&lt;E&gt;</a> | <a href="#">headSet(E toElement)</a>               | Returns a view of the portion of this set whose elements are strictly less than toElement.                           |
| <a href="#">SortedSet&lt;E&gt;</a> | <a href="#">subSet(E fromElement, E toElement)</a> | Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive. |
| <a href="#">SortedSet&lt;E&gt;</a> | <a href="#">tailSet(E fromElement)</a>             | Returns a view of the portion of this set whose elements are greater than or equal to fromElement.                   |

## Abschnitt IV

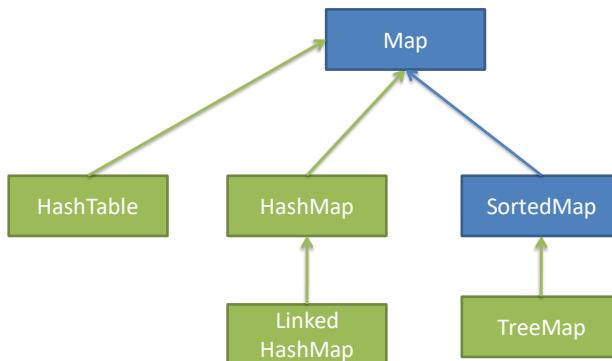
### Maps

## Map

- ▶ Mit einer Map können **Schlüssel / Werte Paare** verwaltet werden
- ▶ Die **Schlüssel** einer Map müssen **eindeutig** sein und sollten **hashCode()** implementieren
  - hashCode() wird aufgerufen, wenn zwei Schlüssel auf Gleichheit überprüft werden sollen
- ▶ Es gibt **sortierte** und **unsortierte**, **geordnete** und **ungeordnete** Maps

## Implementationen

- Anbei einige Beispiele von **Implementationen**:



## Beispiel HashMap & TreeMap

```

public static void main(String[] args) {
    Map<Integer, String> article = new HashMap<Integer, String>();
    Interface Set
    Key Typ
    ValueTyp
    Klasse HashMap

    article.put(1, "Bleistift");
    article.put(20, "A4 Block");
    article.put(3, "Kugelschreiber");

    System.out.println("HashMap sequentiell durchlaufen:");
    print(article);

    // Aus dem HashMap eine TreeMap erzeugen
    Map<Integer, String> articleTree =
        new TreeMap<Integer, String>(article);

    System.out.println("TreeMap sequentiell durchlaufen:");
    print(articleTree);
}
  
```

Konvertierung  
zu TreeMap

## put & get

### ► V **put(K key, V value)**

- Fügt den Schlüssel key und seinen Wert in die Tabelle ein
- Falls ein Schlüssel bereits vorhanden ist, wird sein alter Wert durch den neuen Wert ersetzt
- Die Methode gibt dann den alten Wert des Schlüssels zurück
- Wenn der Schlüssel noch nicht vorhanden ist, gibt die Methode null zurück

### ► V **get(Object key)**

- Gibt das dem Schlüssel zugeordnete Objekt zurück, oder **null**, falls der Schlüssel nicht enthalten ist

## Interface Map (Javadoc)

### Method Summary

`void clear\(\)`

Removes all of the mappings from this map (optional operation).

`boolean containsKey\(Object key\)`

Returns true if this map contains a mapping for the specified key.

`boolean containsValue\(Object value\)`

Returns true if this map maps one or more keys to the specified value.

`Set<entrySet\(\)>`

<`Map.Entry<K,V>`> Returns a `Set` view of the mappings contained in this map.

`boolean equals\(Object o\)`

Compares the specified object with this map for equality.

`V get\(Object key\)`

Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

`int hashCode\(\)`

Returns the hash code value for this map.

`boolean isEmpty\(\)`

Returns true if this map contains no key-value mappings.

## Interface Map II

### Method Summary

|                                                                 |                                                                                         |
|-----------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| <code>Set&lt;K&gt; keySet()</code>                              | Returns a <code>Set</code> view of the keys contained in this map.                      |
| <code>V put(K key, V value)</code>                              | Associates the specified value with the specified key in this map (optional operation). |
| <code>void putAll(Map&lt;? extends K, ? extends V&gt; m)</code> | Copies all of the mappings from the specified map to this map (optional operation).     |
| <code>V remove(Object key)</code>                               | Removes the mapping for a key from this map if it is present (optional operation).      |
| <code>int size()</code>                                         | Returns the number of key-value mappings in this map.                                   |
| <code>Collection&lt;V&gt; values()</code>                       | Returns a <code>Collection</code> view of the values contained in this map.             |

## entrySet

- ▶ Mit `entrySet` ist es auf einfache Art und Weise möglich über die Werte einer Map zu iterieren
- ▶ Dabei erhält bei jedem Durchgang eine Entry mit der man den Schlüssel und den Wert auslesen kann

```
public static void print(Map<Integer, String> map) {
    for (Map.Entry<Integer, String> e : map.entrySet()) {
        Integer key = e.getKey();
        String value = e.getValue();
        System.out.println("key=" + key + ", value=" + value);
    }
}
```

## keySet

- ▶ Alternativ können die Werte auch mit Hilfe des **keySet** durchlaufen werden
  - Hierbei erhält man bei jedem Durchgang den **Schlüssel**
  - Mit diesem kann man via get den dazugehörigen Wert abfragen
- ▶ Ein weitere Möglichkeit besteht mit **values**, welche eine Collection der Werte zurückgibt

```
public static void printByKeySet(Map<Integer, String> map) {
    for (Integer key : map.keySet()) {
        String value = map.get(key);
        System.out.println("key=" + key + ", value=" + value);
    }
}
```



KeySet

Copyright © iten-engineering.ch

Java Einführung

335

335

## Übersicht Collections

| Collection            | ArrayList | Vector | Linked-List | Hash-Set | Tree-Set | Linked-Hash-Set | Hash-Map | Hashtable | Tree-Map | Linked-Hash-Map |
|-----------------------|-----------|--------|-------------|----------|----------|-----------------|----------|-----------|----------|-----------------|
| Eigenschaft           |           |        |             |          |          |                 |          |           |          |                 |
| Geordnet              | ja        | ja     | ja          | nein     | nein     | ja              | nein     | nein      | nein     | ja              |
| Sortiert              | Nein      | nein   | nein        | nein     | ja       | nein            | nein     | nein      | ja       | nein            |
| Null Schlüssel        | n/a       | n/a    | n/a         | n/a      | n/a      | n/a             | ja       | ja        | nein     | Ja              |
| Null Werte            | ja        | ja     | ja          | ja       | nein     | ja              | ja       | ja        | ja       | ja              |
| Indexzugriff          | ja        | ja     | ja          | nein     | nein     | nein            | nein     | nein      | nein     | nein            |
| equals() erforderlich | nein      | nein   | nein        | nein     | ja       | nein            | nein     | nein      | nein     | nein            |
| hashCode()            | nein      | nein   | nein        | ja       | nein     | ja              | ja       | ja        | ja       | ja              |
| Synchronisiert        | nein      | ja     | nein        | nein     | nein     | nein            | nein     | ja        | nein     | nein            |

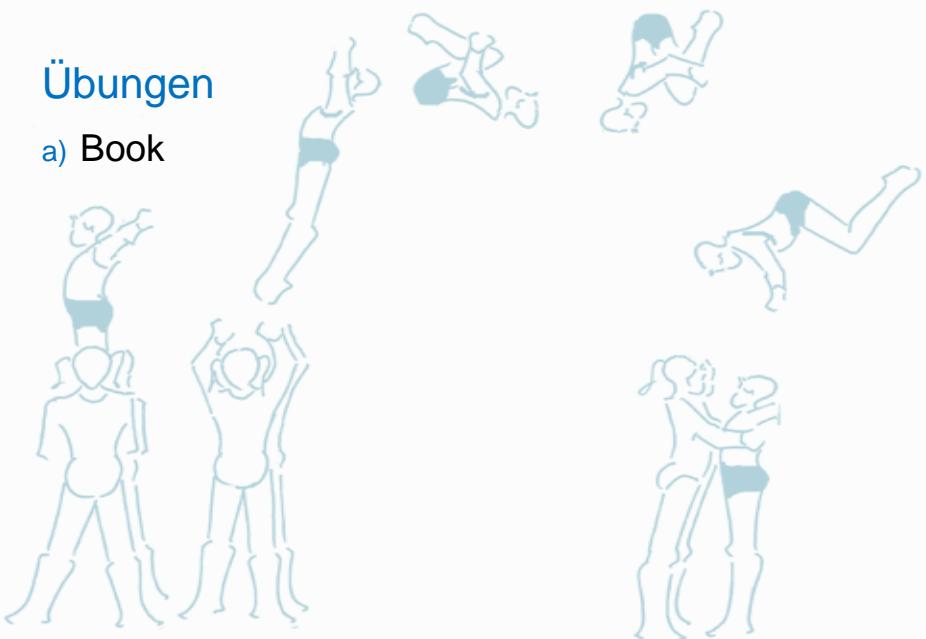
Copyright © iten-engineering.ch

336

336

## Übungen

a) Book



Copyright © iten-engineering.ch

Java Einführung

337

337

## Kapitel 14

# Exceptions

Copyright © iten-engineering.ch

Java Einführung

338

338

## Exception

- ▶ Exception sind Java Klassen zur Erkennung und **Behandlung** von **Ausnahmefällen** wie z.B. Laufzeitfehlern
- ▶ Ein Laufzeitfehler ist ein Fehler, der während der Ausführung eines Programms auftritt und während des Kompilierungsvorganges nicht erkannt werden kann
- ▶ Exception können zum einen von **Java**, als auch durch die **Anwendung selber ausgelöst** werden
- ▶ Die Behandlung erfolgt von Exceptions erfolgt innerhalb von sogenannten **try – catch Blöcken**

## Beispiel try – catch

```
public class TryCatchDemo {
    public static void main(String[] args) {
        try {
            int x = Integer.parseInt("125");
            System.out.println("x = " + x);

            int y = Integer.parseInt("a12");
            System.out.println("y = " + y);
        } catch (NumberFormatException e) {
            System.out.println("Exception = " + e.toString());
        }
    }
}
```

## Exception deklarieren

- ▶ Die Methode Integer.parseInt löst eine NumberFormatException aus, falls der übergebene String nicht in einen Integer konvertiert werden kann
- ▶ Das wird auch aus der Signatur der Methode ersichtlich:

```
public static int parseInt(String s)
    throws NumberFormatException
```

- ▶ Mit der **throws** Klausel definiert die Methode dass Sie eine solche Exception auslösen kann
- ▶ Es ist auch möglich, mehrere Exceptions anzugeben

## Exception deklarieren II

- ▶ In der Javadoc der Methoden findet man dazu jeweils ausführliche Informationen:

```
parseInt

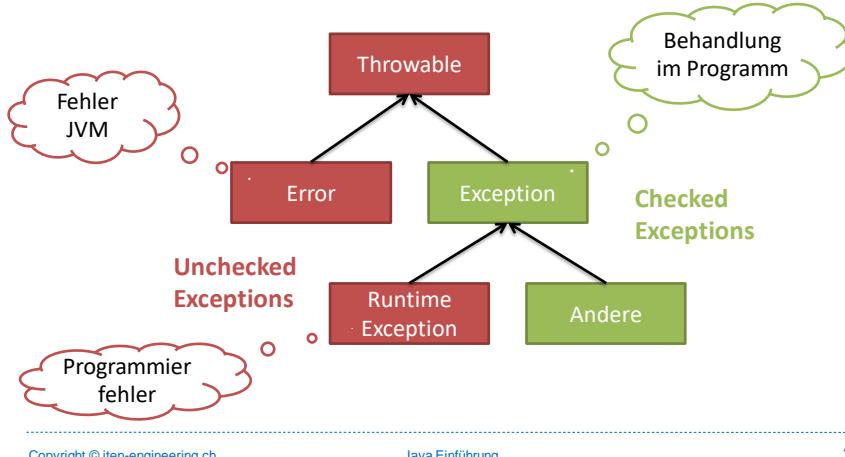
public static int parseInt(String s)
    throws NumberFormatException

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits,
except that the first character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value. The
resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the
parseInt\(java.lang.String, int\) method.

Parameters:
s - a String containing the int representation to be parsed
Returns:
the integer value represented by the argument in decimal.
Throws:
NumberFormatException - if the string does not contain a parsable integer.
```

## Kategorien

- ▶ Java definiert folgende Klassenhierarchie für Ausnahmen:



Copyright © iten-engineering.ch

Java Einführung

343

343

## Throwable

- ▶ Oberklasse aller Exceptions mit zwei wichtigen Methoden
- ▶ `getMessage()`
  - Wenn ein Laufzeitfehler auftritt, generiert die JVM ein Ausnahme Objekt und speichert in ihm eine Textnachricht
  - `getMessage()` gibt diese Nachricht zurück
- ▶ `printStackTrace()`
  - liefert eine Art Momentaufnahme des Programmstacks (stack trace) zum Zeitpunkt des Auftretens des Fehlers in Form einer Textnachricht zurück
  - Damit ist ersichtlich, an welcher Stelle des Programms der Fehler aufgetreten ist

Copyright © iten-engineering.ch

344

344

## Beispiel printStackTrace

The screenshot shows an IDE interface with two main windows. The top window is titled "StackTraceDemo.java" and contains the following Java code:

```

1 package chapter13.exceptions;
2
3 public class StackTraceDemo {
4
5     public static void main(String[] args) {
6
7         try {
8             int x = Integer.parseInt("125");
9             System.out.println("x = " + x);
10
11         int y = Integer.parseInt("a12");
12         System.out.println("y = " + y);
13
14     } catch (NumberFormatException e) {
15         e.printStackTrace();
16     }
17 }
18

```

A red arrow points from a callout bubble labeled "Fehler Zeile :11" to the line of code "int y = Integer.parseInt("a12");". The bottom window is a "Console" tab showing the output of the program execution:

```

<terminated> StackTraceDemo [Java Application] C:\course\java\software\jdk1.6.0_21\bin\javaw.exe (04.06.2011 21:36:51)
x = 125
java.lang.NumberFormatException: For input string: "a12"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
        at java.lang.Integer.parseInt(Integer.java:449)
        at java.lang.Integer.parseInt(Integer.java:499)
        at chapter13.exceptions.StackTraceDemo.main(StackTraceDemo.java:11)

```

Copyright © iten-engineering.ch      Java Einführung      345

345

## Error

- ▶ Schwerwiegende Fehler, nach deren Auftreten die Fortsetzung des Programms nicht sinnvoll bzw. gar nicht möglich ist
- ▶ Führen zu einem Programmabbruch und zur Beendigung des Interpreters
- ▶ Beispiele:
  - Fehler der JVM
  - Speicherüberlauf (Out of Memory)

346

## RuntimeException

- ▶ Werden meistens durch Programmierfehler ausgelöst, d.h. könnte vermieden werden
- ▶ Beispiele:
  - `ArrayIndexOutOfBoundsException`
  - `NegativeArraySizeException`
  - `StringIndexOutOfBoundsException`
- ▶ Ein häufiger Fehler ist der Gebrauch einer Referenz die anstatt auf ein gültiges Objekt zu zeigen null ist
- ▶ Versucht man nun mit einer solchen Referenz eine Methode aufzurufen, so führt dies zu einer `NullPointerException`

## Exception

- ▶ Exception und alle davon abgeleiteten Klassen sind sogenannte **Checked Exceptions**
- ▶ Sie repräsentieren Systemfehler die in der Anwendung behandelt werden können, ohne das das Programm abgebrochen werden muss
- ▶ Beispiele:
  - `FileNotFoundException`
  - `IOException`
- ▶ Checked Exceptions müssen **abgefangen** werden **oder** mittels throws Deklaration bei der Methoden **deklariert** sein
  - Im Gegensatz dazu müssen Unchecked Exceptions nicht zwingend behandelt oder angegeben werden

## Exception auslösen

- ▶ Exception kann man an einer beliebigen Stelle im Programm mit `throw` auch selber auslösen:

```
public class ThrowsDemo {
    public static void print(String message) {
        if (message == null) {
            throw new IllegalArgumentException
                ("Der Parameter message darf nicht null sein.");
        }
        System.out.println(message);
    }
    public static void main(String[] args) {
        try {
            print("Message A");
            print(null);
            print("Message B");
        } catch (Exception e) {
            System.out.println(e.toString());
        }
    }
}
```

Exception erstellen und auslösen

```
Problems @ Javadoc Declaration Console Search
<terminated> > ThrowsDemo [Java Application] C:\course\java\software\jdk1.6.0_21\bin\javaw.exe (04.06.2011 22:31:04)
Message A
java.lang.IllegalArgumentException: Der Parameter message darf nicht null sein.
```

## Exception weitergeben

- ▶ Exception können auch weitergegeben werden
- ▶ Entweder fängt man Sie gar nicht ab, dann wird Sie automatisch an den nächsten Aufrufer weiter propagiert
- ▶ Oder man fängt eine Exception ab, führt eine Aktion aus und wirft Sie weiter
- ▶ Oder man fängt eine Exception ab, führt die gewünschten Aktionen aus, erstellt eine neue (andere) Exception und wirft diese weiter
  - Dieser Fall tritt häufig auf, wenn man technische Exceptions in benutzerfreundliche umwandelt

## Mehrere Exception behandeln

- Man kann auch mehrere Exceptions separat behandeln, indem man für jede mögliche Exception einen eigenen catch Block erstellt

```
public void printFile(String fileName){           // try to read and print the given file
    BufferedReader input = null;

    try {
        input = new BufferedReader( new FileReader(fileName) );
        String line = null;
        while (( line = input.readLine()) != null) {
            System.out.println(line);
        }
    } catch (FileNotFoundException ex) {
        ...
    } catch (IOException ex){
        ...
    }
}
```

Separate  
Fehlerbehandlung

## Multi catch

- Seit Java 7 besteht auch die Möglichkeit mehrere Exceptions in einem catch Block zu behandeln (**multi catch**)
- Das ist praktisch wenn das **Fehlerhandling** für mehrere Ausnahmefälle **gleich** ist

```
try {
    ...do something...

} catch (FileNotFoundException | IOException ex) {
    ...
}
```

## Abschlussarbeiten mit finally

- ▶ Möchte man in einer Methode eine Teil Programmcode in jeden Fall ausführen, **unabhängig ob ein Exception ausgelöst wurde oder nicht**, so kann man die try-catch Struktur mit einem **finally** Block erweitern
- ▶ Es ist auch möglich einen try-finally Block ohne catch zu definieren
- ▶ Finally Blöcke braucht man häufig für die **Freigabe von Ressourcen**, so dass diese auch im Fehlerfall korrekt „aufgeräumt“ werden

## Beispiel try – finally

```
public class FinallyDemo {

    public static void print(String message) throws InterruptedException {
        long startTime = System.currentTimeMillis();
        try {
            if (message == null) {
                Thread.currentThread().sleep(3);
                throw new IllegalArgumentException();
            }
            System.out.println(message);
        } finally {
            long endTime = System.currentTimeMillis() - startTime;
            System.out.println("print used " + endTime + " ms");
        }
    }

    public static void main(String[] args) {
        try {
            print("Message A");
            print(null);
            print("Message B");
        } catch (Exception e) {
            System.out.println(e.toString());
        }
    }
}
```

The screenshot shows the Java IDE interface with the code editor and a terminal window. The terminal displays the following output:

```
Message A
print used 0 ms
print used 15 ms
java.lang.IllegalArgumentException
```

## Eigene Exception

- ▶ Möchte man eine eigene Exception erstellen, so leitet man diese i.d.R. von java.lang.Exception oder einer Unterklasse davon ab
- ▶ Möchte man eine Unchecked Exception, so muss man seine Klasse von der java.lang.Runtime Exception oder einer Unterklasse davon ableiten
- ▶ Die Klasse sollte mindestens
  - einen Standard Konstruktor (keine Parameter)
  - sowie einen Konstruktor mit einem String Parameter implementieren

## Beispiel eigene Exception

```
public class CustomException extends Exception {
    int errorNumber;

    public CustomException() {
        super();
    }

    public CustomException(String message) {
        super(message);
    }

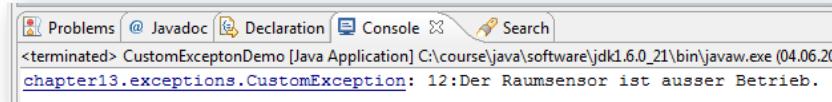
    public CustomException(int errorNumber, String message) {
        super(message);
        this.errorNumber = errorNumber;
    }

    @Override
    public String getMessage() {
        return errorNumber + ":" + super.getMessage();
    }
}
```

## Beispiel eigene Exception II

```
public class CustomExceptionDemo {
    public static void checkHeat() throws CustomException {
        // ...
        throw new CustomException(12, "Der Raumsensor ist ausser Betrieb.");
    }

    public static void main(String[] args) {
        try {
            checkHeat();
        } catch (CustomException e) {
            System.out.println(e.toString());
        }
    }
}
```



## try-with-resources

- ▶ Seit **Java 7** gibt es das sogenannte **try-with-resources**
- ▶ Damit können Resourcen direkt im try geöffnet werden
- ▶ Nach dem try Block werden diese Resourcen durch Java automatisch wieder geschlossen
  - Dies wird auch im Fehlerfall (bei einer Exception) gemacht
- ▶ Dadurch wird die Programmierung noch **sicherer**, da Resourcen **automatisch** wieder **freigehen** werden, ohne das dies explizit ausprogrammiert werden muss

## Beispiel try-with-resources

```

public static void main(String[] args) {

    String fileName = TestData.getAbsoluteFilename("file-writer-demo.txt");

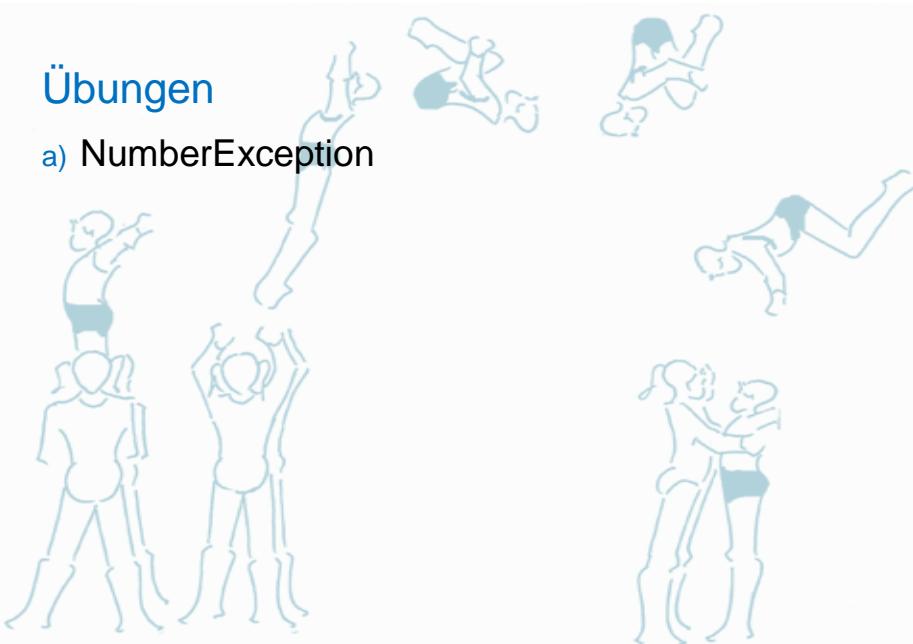
    // try with resources, the writer will be automatically closed at the end
    try (FileWriter writer = new FileWriter(fileName)) {
        writer.write("1. Zeile: Test zur Ausgabe \r\n");
        writer.write("2. Zeile: in eine Datei.");
    } catch (IOException e) {
        System.out.println("Demo failed with: " + e.toString());
    }
}

```

- ▶ Try-with-resources geht für alle Resourcen, die das **Closable** Interface implementieren

## Übungen

- a) NumberException



## Kapitel 15

### Assertions und Annotations

---

Copyright © iten-engineering.ch

Java Einführung

361

361

## Abschnitt I

### Assertions

---

Copyright © iten-engineering.ch

Java Einführung

362

362

## Assertions

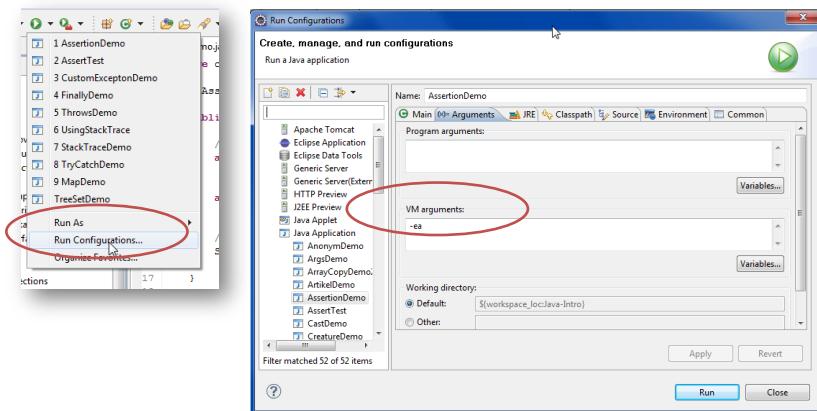
- ▶ Mit Assertions (**Behauptungen**) können Programmzustände geprüft und überwacht werden
- ▶ Assertions sind **per Default ausgeschaltet** und werden nicht überwacht
- ▶ Mit der **Option –ea** kann der JVM mitgeteilt werden, dass die Assertions ausgewertet werden sollen
- ▶ Assertions können so zum Beispiel während der Entwicklung gezielt eingeschaltet werden, um fehlerhafte Zustände möglichst rasch zu entdecken
- ▶ Da Assertions nicht zwingend ausgewertet werden müssen, sie **nicht geeignet**, um einen **korrekten Programmablauf** sicherzustellen

## Syntax

- ▶ **assert expression : message;**
  - Die Behauptung beginnt mit dem **Schlüsselwort assert**
  - Anschliessend folgt ein **boolscher Ausdruck**
    - Ist das Ergebnis true, wird die nächste Anweisung ausgeführt
    - Ist das Ergebnis **false**, wird ein **AssertionError** ausgelöst
  - Im dritten Teil folgt eine **String Meldung**, die dem **AssertionError** als Message mitgegeben wird. Dieser Teil ist optional
- ▶ Damit die Behauptung während der Ausführung überprüft wird, müssen die Assertions beim Start der Anwendung aktiviert werden

## Enable Assertions in Eclipse

- Im Register Arguments der Run Configuration kann die VM Option –ea gesetzt werden:



Copyright © iten-engineering.ch

Java Einführung

365

365

## Beispiel assert

```
class AssertionDemo {

    public static void setRoomTemp(double temp) {

        // Soll Temperatur auf "normalen Bereich" prüfen
        assert temp > 17.0 : "Die Raumtemperatur ist mit " + temp
            + " Grad zu niedrig.";

        Wenn die Behauptung wahr ist, wird das Programm weiter ausgeführt.

        assert temp < 22.0 : "Die Raumtemperatur ist mit " + temp
            + " Grad zu hoch.";

        // Heizung steuern
        System.out.println ("Die Solltemperatur wird auf " + temp
            + " Grad eingestellt.");
    }
}
```

Copyright © iten-engineering.ch

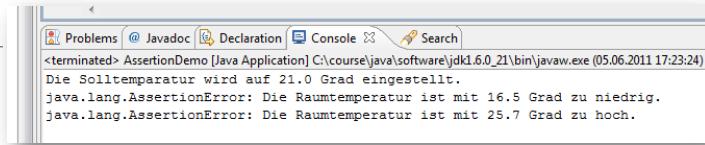
Java Einführung

366

366

## Beispiel assert II

```
class AssertionDemo {  
  
    public static void setRoomTemp(double temp) {...}  
  
    public static void main(String args[]) {  
        setRoomTemp(21.0);  
        try {  
            setRoomTemp(16.5);  
        } catch (AssertionError e) {  
            System.out.println(e.toString());  
        }  
        try {  
            setRoomTemp(25.7);  
        } catch (AssertionError e) {  
            System.out.println(e.toString());  
        }  
    }  
}
```



## Abschnitt II

### Annotations

## Annotationen

- ▶ Mit Java 5 wurden Annotationen eingeführt
- ▶ Damit ist es möglich auf einem Java Element (Feld, Constructor, Methode, Klasse, etc.) Metainformationen zu hinterlegen
- ▶ Die Annotationen haben keinen direkten Einfluss auf den Code
- ▶ Je nach Annotation ist der Anwendungszweck und Zeitpunkt der Auswertung unterschiedlich
- ▶ Es gibt zum Beispiel Annotationen für den Compiler, von Frameworks, Tools oder Javadoc

## Verfügbarkeit

- ▶ Mit der sogenannten Retention Policy wird angegeben wann eine Annotation im System verfügbar ist
- ▶ Dabei kann man auswählen zwischen SOURCE, CLASS und RUNTIME
  - Eine nur im Sourcecode vorhandenen Annotation wird zum Beispiel von JavaDoc für das generierten der Java Hilfe verwendet
  - Eine in der Class Datei vorhandene Annotation kann zum Beispiel von einen Java EE Server zum Deploy Zeitpunkt ausgewertet werden
  - Annotations welche zur Laufzeit (Runtime) verfügbar sind können im laufenden Programm via Reflexion abgefragt werden

## Vordefinierte Annotationen

- Im Package `java.lang` werden die u.a. die folgenden Annotationen für den Compiler definiert:

| Annotation                     | Verwendung                                                                                                                                                                                         |
|--------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>@Deprecated</code>       | Markierung von veralteten (nicht mehr zu verwendenden) Elementen wie Methoden oder Klassen                                                                                                         |
| <code>@Override</code>         | Kennzeichnet eine Methode die eine Methode einer Oberklasse überschreibt oder eine Methode eines Interface implementiert                                                                           |
| <code>@SuppressWarnings</code> | Unterdrücken von Compiler Warnings die dieser sonst generieren würde                                                                                                                               |
| <code>@SafeVarargs</code>      | Kann bei Methoden und Konstruktoren mit variabler Anzahl Argumenten verwendet werden um sicherzustellen dass der Code keine potentiell unsicheren Operationen (mit den varags Parametern) ausführt |

## Beispiele

```
public interface NumUtil {

    /**
     * Gets the minimum value of the range of values.
     *
     * @param values A variable number of values.
     * @return The minimum value.
     */
    public Integer min(Integer... values);

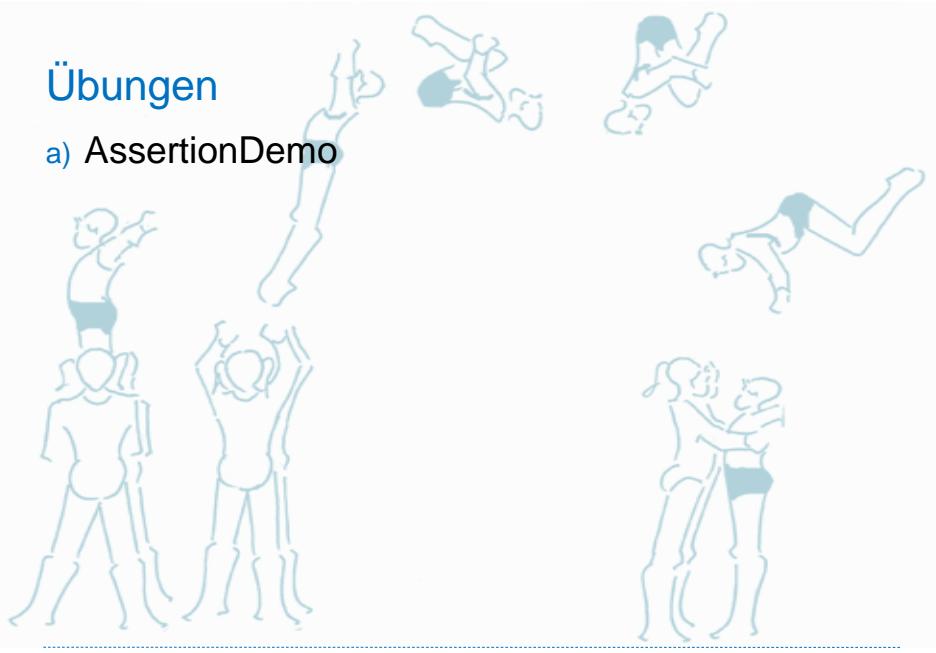
}
```

```
public class NumUtilImpl implements NumUtil {

    @Override
    public Integer min(Integer... values) {
        ...
        ...
    }
}
```

## Übungen

### a) AssertionDemo



Copyright © iten-engineering.ch

Java Einführung

373

373

## Kapitel 16

### Dateien

Copyright © iten-engineering.ch

Java Einführung

374

374

## Übersicht

- ▶ Bis **Java 6** verwendete man für das Arbeiten mit Dateien die Klassen des Package `java.io`
- ▶ Ab **Java 7** steht mit dem Package `java.nio` eine neue Lösung zur Verfügung
  - Die neuen Klassen bieten **mehr Möglichkeiten** als die bisherigen und setzen das **Exception Handling Konzept** von Java konsequent um
  - Für die Dateiverwaltung stehen u.a. die Klassen `FileSystem`, `Path` und `Files` zur Verfügung
  - Für die **Interoperabilität** mit den neuen Klassen bietet die bisherige Klasse `File` die Methode `toPath()`, welche ein `Path` Objekt zurück gibt
- ▶ Mit **Java 8** gab es im Rahmen der Lambda Expressions weitere Ergänzungen um `Files` mit functional Streams bearbeiten zu können

## Abschnitt I

**java.nio**

## FileSystems

- ▶ Die Klasse `FileSystems` definiert `Factory` Methoden für die Erstellung von Objekten des Type `FileSystem`
- ▶ Enthält die Methode `getDefault()` welche das Default `FileSystem` zurückliefert, welches der Java Virtual Machine zur Verfügung steht
- ▶ Liefert weitere `Factory` Methoden für die Erstellung von `FileSystem` Instanzen via URI

## FileSystem

- ▶ Bildet die `Schnittstelle` zum File System
- ▶ Bietet `Factory` Methoden zur Erstellung von Objekten des File System wie z.B. `Path`
- ▶ Definiert eine alternative Möglichkeit für den erhalten des File Separator der aktuellen Betriebssystem
  - Microsoft Betriebssysteme verwenden den Backslash «\» als Separator
  - Unix und Linux Systeme verwenden ein Slash «/»

## FileSystem – Beispiel Separator

```

import java.io.File;
import java.nio.file.FileSystems;

public class FileSeparatorDemo {

    public static void main(String[] args) {
        System.out.println("File Separator:");
        // file separator
        System.out.println("File.separator = " + File.separator);

        // file separator mit Java 7
        System.out.println("FileSystems.getDefault().getSeparator() = " +
                           FileSystems.getDefault().getSeparator());
    }
}

```

## Path

- ▶ Mit dem Path Objekt können **Dateien** in einem File System **lokalisiert** werden
- ▶ Path Objekte können mit der **Factory Klasse Paths** erzeugt werden
- ▶ Ein Path Objekt ist immer **Systemabhängig**, je nachdem auf welcher Plattform das Programm ausgeführt wird
- ▶ Mit Path können die **Eigenschaften** einer Datei oder eines **Verzeichnis** abgefragt werden

## Path – Beispiel

```
Path path = Paths.get("src/chapter16/files/nio/demo");
System.out.format("getFileName: %s%n", path.getFileName());
System.out.format("getFileSystem: %s%n", path.getFileSystem());
System.out.format("getName(1): %s%n", path.getName(1));
System.out.format("getNameCount: %d%n", path.getNameCount());
System.out.format("getRoot: %s%n", path.getRoot());
System.out.format("getParent: %s%n", path.getParent());
System.out.format("isAbsolute: %s%n", path.isAbsolute());
System.out.format("toString: %s%n", path.toString());
System.out.println();
```

getFileName: demo  
 getFileSystem: sun.nio.fs.WindowsFileSystem@5d86aad9  
 getName(1): chapter16  
 getNameCount: 5  
 getRoot: null  
 getParent: src\chapter16\files\nio  
 isAbsolute: false  
 toString: src\chapter16\files\nio\demo



## Files

- ▶ Die Klasse Files besteht ausschliesslich aus **statischen Methoden**
- ▶ Mit Files können **Verzeichnisse und Dateien erstellt, gelesen oder mutiert** werden
- ▶ Für das Ausführen der Methoden werden **Instanzen** der Klasse **Path** verwendet
- ▶ Bei Verwendung von Ressourcen welche **java.io.Closable** implementieren ist zu beachten, dass diese am Ende der Verarbeitung (auch im Fehlerfall) korrekt geschlossen werden
  - am einfachsten ist dabei die Ausführung innerhalb eines **try-with-resources** Block (seit Java 7)
  - oder mit Hilfe eines **try-catch-finally** Block

## Files – Beispiel Verzeichnis erstellen

```

Path dir = Paths.get(DIR_NAME);

if (Files.exists(dir)) {
    System.out.println("Das Verzeichnis " + dir.toString()
        + " existiert bereits.");
}

} else {
    System.out.println("Das Verzeichnis " + dir.toString()
        + " wird neu erstellt.");

    try {
        Files.createDirectory(dir); ←
    } catch (IOException e) {
        System.out.println("> IOException bei der Verzeichnis-Erstellung:"
            + e.toString());
        System.exit(-1);
    }
}

```

## Files – Beispiel Datei erstellen

```

Path file = Paths.get(FILE_NAME);

if (Files.exists(file)) {
    System.out.println("Die Datei " + file.getFileName()
        + " existiert bereits.");
}

} else {
    System.out.println("Die Datei " + file.toString()
        + " wird neu erstellt.");

    try {
        Files.createFile(file); ←
    } catch (IOException e) {
        System.out.println("> IOException bei der Datei-Erstellung: "
            + e.toString());
        System.exit(-2);
    }
}

```

## SeekableByteChannel

- ▶ Für den **wahlfreien Zugriff** auf eine Datei wird mit Java 7 das Interface SeekableByteChannel zur Verfügung gestellt
- ▶ Dabei kann mit Hilfe eines **Pointer** auf eine beliebige Position innerhalb einer Datei gezeigt werden
- ▶ Ausgehend von dieser **aktuellen Position** können nun **Daten gelesen oder geschrieben** werden
- ▶ Für den Zugriff auf Dateien implementiert die Klasse **FileChannel** das genannte Interface

## FileChannel – Beispiel write

```

try (FileChannel fc = (FileChannel.open(file, READ, WRITE))) {
    // reset position
    fc.position(0);
    // write
    byte data[] = "AAAAA\nBBBBB\nCCCCC\n".getBytes();
    ByteBuffer wbuf = ByteBuffer.wrap(data);
    fc.write(wbuf);

    System.out.println("Daten geschrieben, Dateigrösse=" + fc.size());
} catch (IOException e) {
    System.out.println("IOException bei der Demo: " + e.toString());
    System.exit(-1);
}

```



## FileChannel – Beispiel read

```

try (FileChannel fc = (FileChannel.open(file, READ, WRITE))) {
    // reset position
    fc.position(0);

    // read
    ByteBuffer rbuf = ByteBuffer.allocate(12);
    fc.read(rbuf);

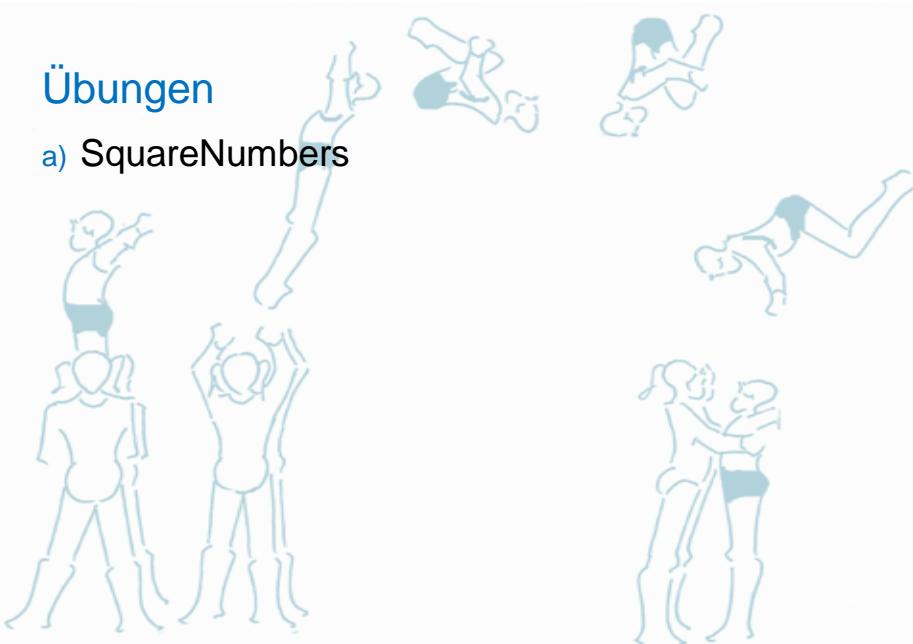
    System.out.println();
    System.out.println("Ausgabe Testdaten:");
    System.out.println(rbuf.toString());
    System.out.println(new String(rbuf.array()));

} catch (IOException e) {
    System.out.println("IOException bei der Demo: " + e.toString());
    System.exit(-2);
}

```

## Übungen

### a) SquareNumbers



## Abschnitt II

### java.io

## File

- ▶ Die Bearbeitung von Dateien und Verzeichnissen erfolgt in Java mit der Klasse **java.io.File**
- ▶ Das File Objekt **repräsentiert** den **Namen** einer **Datei** oder eines **Verzeichnisses**
  - Die Datei / das Verzeichnis muss dabei physikalisch nicht existieren
- ▶ File stellt zahlreiche Methoden zur **Navigation** im **Dateisystem** und zur **Manipulation** von Dateien und Verzeichnissen zur Verfügung

## File – Konstruktoren (Javadoc)

- ▶ Die Erstellung einer File Instanz
  - Setzt die Existenz der Datei / des Verzeichnisses nicht voraus
  - Bedeutet nicht, dass die Datei / das Verzeichnis angelegt wird
- ▶ Das File Objekt ist lediglich eine Beschreibung einer Datei / eines Verzeichnisses

### Constructor Summary

**File(File parent, String child)**

Creates a new File instance from a parent abstract pathname and a child pathname string.

**File(String pathname)**

Creates a new File instance by converting the given pathname string into an abstract pathname.

**File(String parent, String child)**

Creates a new File instance from a parent pathname string and a child pathname string.

**File(URI uri)**

Creates a new File instance by converting the given file: URI into an abstract pathname.

Copyright © iten-engineering.ch

Java Einführung

391

391

## File – Beispiele Konstruktoren

```
// absoluter Pfad eines Ordners (Windows-Syntax)
File f1 = new File("c:\\\\data");

// absoluter Pfad einer Datei (Windows-Syntax)
File f2 = new File("c:\\\\data\\\\info.txt");

// relativer Pfad einer Datei (Windows-Syntax)
File f3 = new File("../\\\\data\\\\info.txt");

// relativer Pfad einer Datei (Unix-Syntax)
File f4 = new File("../..\\\\info.txt");

// relativer Pfad einer Datei mit Separator
File f5 = new File("data" + File.separator + "info.txt");

// URI einer Datei
URI uri = new URI("http://java.sun.com/demo/SwingSet2.java");
File f6 = new File(uri);
```

The code examples illustrate various ways to create File objects. Annotations explain specific aspects:

- An annotation points to the first line: "Backslash «\\» wird in einem String wird in Java ausmaskiert" (Backslash «\\» is masked in a string).
- An annotation points to the third line: "File.separator enthält das Trennzeichen je jeweiligen Betriebssystems." (File.separator contains the separator character for the respective operating system).
- An annotation points to the last line: "Kann eine URISyntaxException auslösen" (Can cause a URISyntaxException).

Copyright © iten-engineering.ch

Java Einführung

392

392

## File – Methoden

- ▶ File bietet eine Vielzahl von Methoden
  - für die Abfrage von Pfad und Dateiinformationen
  - die Erstellung von Dateien und Verzeichnissen
  - die Umbenennung oder das Löschen von Dateien und Verzeichnissen
- ▶ Mit Java 6 wurden weitere Methoden hinzugefügt
  - für die Abfrage des belegten und freien Diskspace
  - das Setzen von Schreib- und Leserechten
- ▶ Detaillierte Informationen zu den einzelnen Methoden findet man wie immer in der Javadoc

## File – Beispiel Directory

```
// relativer Pfad zum Package chapter15.files.data
// im Eclipse Projekt definieren
String dirName = "src" + File.separator + "chapter15" + File.separator
  + "files" + File.separator + "data";

// neues File Objekt mit den gewünschten Verzeichnis erstellen
File dir = new File(dirName);

if (dir.exists()) {
    System.out.println
        ("Verzeichnis " + dir.getPath() + " existiert bereits.");
} else {
    dir.mkdir();
    System.out.println
        ("Verzeichnis " + dir.getPath() + " wurde neu erstellt.");
}
```

## File – Beispiel Datei anlegen

```
// Datei Name (inkl. Pfad) erstellen
String fileName = dirName + File.separator + "infos.txt";

// neues File Objekt mit den gewünschten Dateinamen erstellen
File file = new File(fileName);

if (file.exists()) {
    System.out.println("Datei " + file.getPath() + " existiert bereits.");
} else {
    try {
        file.createNewFile();
        System.out.println("Datei " + file.getPath() + " wurde neu erstellt.");
    } catch (IOException ex) {
        System.out.println("Failed with IOException: " + ex.toString());
        return;
    }
}
}

Abfrage ob Datei bereits existiert
Datei anlegen
```

Copyright © iten-engineering.ch

Java Einführung

395

395

## File - Beispiel Liste

```
public static void listFiles(File directory, int ident) {
    String[] list = directory.list();
    for (String fileName : list) {
        for (int i = 0; i < ident; i++) {
            System.out.print("  ");
        }
        File child = new File(directory, fileName);
        if (child.isDirectory()) {
            System.out.println(fileName + ":");
            listFiles(child, ++ident);
            ident--;
        } else {
            System.out.println(fileName);
        }
    }
}

Liste aller Datei und Verzeichnis Namen
Mit dem Namen ein neues File Objekt erstellen
Bei Verzeichnis rekursiver Aufruf, sonst Ausgabe
```

Copyright © iten-engineering.ch

Java Einführung

396

396

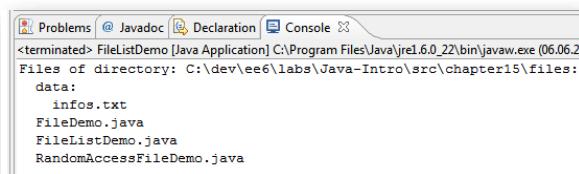
## File – Beispiel Liste II

```
public static void main(String[] args) {

    // create file
    String dirName = "src" + File.separator + "chapter15"
        + File.separator + "files";

    File file = new File(dirName);

    // list files
    System.out.println(
        ("Files of directory: " + file.getAbsolutePath() + ":");
        listFiles(file, 1);
    }
}
```



## RandomAccessFile

- ▶ Für die wahlfreien Zugriff auf Dateien stellt Java die Klasse `java.io.RandomAccessFile` zur Verfügung
- ▶ Mit Hilfe eines **Zeigers** kann auf eine **beliebige Position** innerhalb der aktuellen Datei verweist werden
- ▶ Damit sind, an einer beliebigen Stelle innerhalb der Datei, **lesende und schreibende Zugriffe** möglich

Datei:



## RandomAccessFile – Konstruktoren

- ▶ Bei der Erstellung eines Objektes gibt man mit dem **mode** an, ob man auf die Datei **lesend** oder **lesend und schreibend** zugreifen will
- ▶ Der **Dateizeiger** gibt an ab wo die Daten gelesen oder geschrieben werden soll und wird am Anfang automatisch auf die **Position 0** gesetzt

### Constructor Summary

[RandomAccessFile\(File file, String mode\)](#)

Creates a random access file stream to read from, and optionally to write to, the file specified by the [File](#) argument.

[RandomAccessFile\(String name, String mode\)](#)

Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

## RandomAccessFile – Zugriff Modi

- ▶ Die mögliche Werte vom Parameter **mode** sind in der folgenden Tabelle ersichtlich:

| Value | Meaning                                                                                                                                                                     |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| "r"   | Open for reading only. Invoking any of the write methods of the resulting object will cause an <a href="#">IOException</a> to be thrown.                                    |
| "rw"  | Open for reading and writing. If the file does not already exist then an attempt will be made to create it.                                                                 |
| "rws" | Open for reading and writing, as with "rw", and also require that every update to the file's content or metadata be written synchronously to the underlying storage device. |
| "rwd" | Open for reading and writing, as with "rw", and also require that every update to the file's content be written synchronously to the underlying storage device.             |

## RandomAccessFile

### Beispiel Datei erstellen & schliessen

```

RandomAccessFile file = null;

try {
    // File erstellen
    String dirName = "src" + File.separator + "chapter15"
        + File.separator + "files" + File.separator + "data";

    File fileName = new File(dirName + File.separator + "test.dat");

    // RandomAccessFile erstellen
    file = new RandomAccessFile(fileName, "rw");
    ...
} catch (Exception e) {
    ...
} finally {
    if (file != null) {
        // Datei schliessen
        try {
            file.close();
        } catch (Exception e) {
            // ignore
        }
    }
}

```

## RandomAccessFile – Daten schreiben

- ▶ Zum schreiben in eine Datei steht für jeden primitiven Datentyp eine write Methode zur Verfügung, wie zum Beispiel
  - `writeBoolean` für boolean Typen
  - `writeBytes` für bytes
  - etc
- ▶ Für String kann man die zwischen den beiden folgenden Methoden wählen:
  - `writeBytes` speichert einen String als eine Reihenfolgen von 1 Byte Zeichen
  - `writeChars` erzeugt eine Folge von Unicode Zeichen (je 2 Byte)

## RandomAccessFile – Daten lesen

- ▶ Das lesen ab der aktuellen Position erfolgt mit einer, dem Datentyp entsprechenden read Methode, wie zum Beispiel
  - boolean `readBoolean()` für boolsche Typen
  - etc
- ▶ Mit `void readFully(byte[] b)` können mehrere Bytes (`b.length`) in ein Byte Array gelesen werden
- ▶ Mit `String readLine()` werden alle Zeichen bis zum Zeilenende als String zurückgeliefert
- ▶ Alle Methoden erhöhen jeweils die Zeigerposition um die Anzahl der gelesenen Bytes

## RandomAccessFile – Zeiger

- ▶ Der Datei Zeiger gibt die Position an, ab der die Daten gelesen oder geschrieben werden sollen
- ▶ Am Anfang wird der Zeiger auf die Position 0 gesetzt
- ▶ Mit der Methode `getFilePointer()` kann die aktuelle Positon abgefragt werden
  - Die Methode gibt einen long Wert mit dem Offset zur Startposition zurück
- ▶ Mit der Methode `seek(long position)` wird der Zeiger zur angegeben Position verschoben
- ▶ Mit `skipBytes(int n)` wird die Zeiger Position relativ um n Bytes verschoben

## RandomAccessFile

### Beispiel Daten schreiben & lesen

```
// Datei schreiben
file.writeBytes("AAAAAA\nBBBBBB\nCCCCCC\n");
System.out.println("Laenge der Datei:\n" + file.length());

// Laenge des Files wird auf 12 reduziert
file.setLength(12);
System.out.println("Laenge der Datei nach Reduktion:\n" + file.length());

// Inhalt ausgeben
file.seek(0);
System.out.println("Ausgabe der Testdaten nach erstem Mal lesen:");

for (int i = 1; i <= 2; i++) { // 2 Zeilen lesen
    System.out.println(file.readLine());
}
```

Copyright © iten-engineering.ch

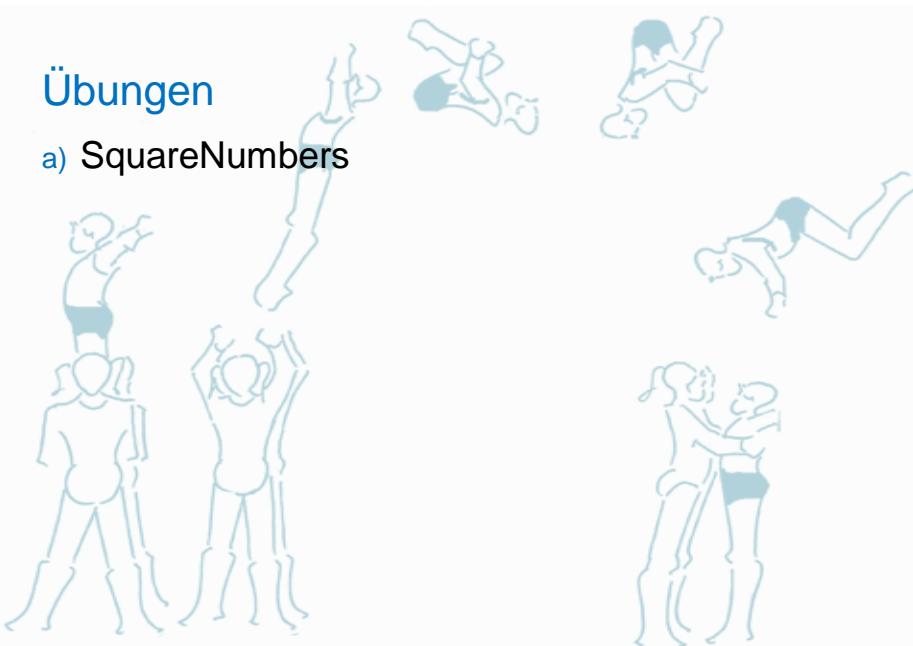
Java Einführung

405

405

## Übungen

### a) SquareNumbers



Copyright © iten-engineering.ch

Java Einführung

406

406

## Kapitel 17

### Streams

407

## Abschnitt I

### Einleitung

408

## Streams

- ▶ Für die **sequentielle Ein- und Ausgabe** von Daten werden in Java Streams verwendet
- ▶ Mit einem **Input Stream** können Daten aus einer beliebigen Quelle (Tastatur, Datei, etc.) gelesen werden
- ▶ Mit einem **Output Stream** können Daten in ein beliebiges Ziel (Bildschirm, Datei, etc.) geschrieben werden
- ▶ Man unterscheidet zwischen **Character** und **Byte Streams**

## Stream Typen

- ▶ **Character Streams**
  - sind zeichenorientiert
  - transportieren Unicode Zeichen
  - mit einer Transportbreite von jeweils 16 Bit
  - sind für die Ein- und Ausgabe von Texten
- ▶ **Byte Streams**
  - sind byteorientiert
  - Transportieren jeweils 8 Bit
  - sind für die Ein- und Ausgabe von Daten im Byte Format, wie zum Beispiel Bilder, Programme, etc.

## Superklassen & Interfaces (Package java.io)

- ▶ Die Superklassen sind jeweils abstrakt
- ▶ Je nach Typ, werden verschiedene Interface implementiert
  - Closable zum Schliessen von Ein- / Ausgabe Streams
  - Flushable zum Leeren von Ausgabe Streams
  - Readable zum Lesen von Charakter Streams
  - Appendable zum Anfügen an Character Streams

| Typ              | Breite | Eingabe     |                    | Ausgabe      |                                 |
|------------------|--------|-------------|--------------------|--------------|---------------------------------|
|                  |        | Superklasse | Interface          | Superklasse  | Interface                       |
| Character Stream | 16 Bit | Reader      | Closable, Readable | Writer       | Closable, Flushable, Appendable |
| Byte Stream      | 8 Bit  | InputStream | Closable           | OutputStream | Closable, Flushable             |

## Fehlerbehandlung

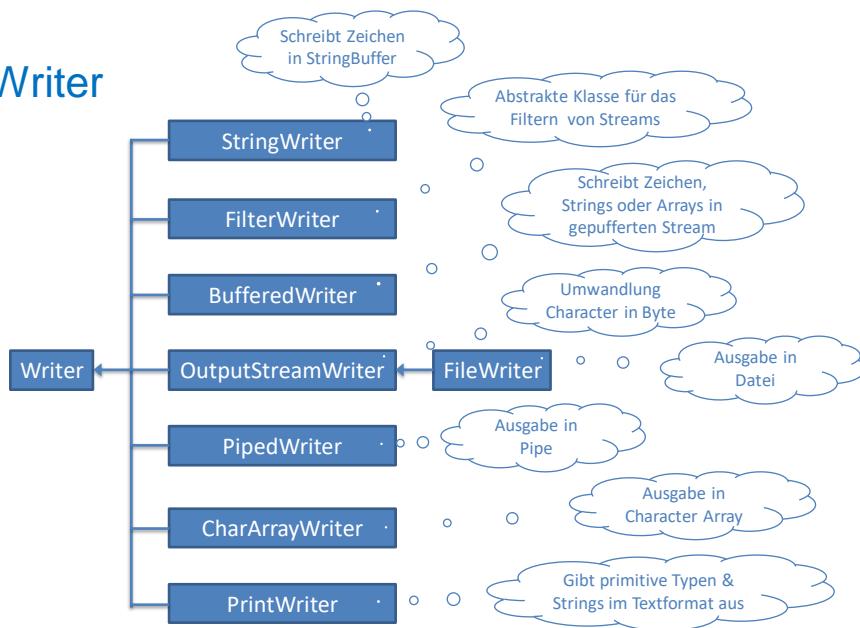
- ▶ Bei der Arbeit mit Streams ist darauf zu achten, dass alle geöffneten Ressourcen am Ende der Bearbeitung wieder geschlossen werden
- ▶ Seit Java 7 steht dabei mit dem **try-with-resources** eine vereinfachte Variante zur Verfügung,
  - dabei wird eine Ressource die **java.io.Closable** implementiert am Ende des try automatisch von der Java Virtual Machine geschlossen wird
- ▶ Alternativ kann das Schliessen der Ressourcen auch wie bisher explizit erfolgen

## Abschnitt II

### Charakter Streams

413

## Writer



414

## FileWriter

- ▶ Mit dem FileWriter werden **Daten** in eine **Datei** ausgegeben
- ▶ Bei der Erstellung des FileWriter kann man angeben, ob die Datei **neu** erstellt werden sollen oder ob die Daten an die bestehenden **anhängt** (append) werden sollen
- ▶ Für die Ausgabe stehen anschliessend verschiedene **write** Methoden zur Verfügung
- ▶ Am Ende wird der Stream mit der **close()** Methode geschlossen

## Beispiel FileWriter

```
public static void main(String[] args) {
    FileWriter fw = null;

    try {
        fw = new FileWriter("file-writer-demo.txt");

        fw.write("1. Zeile: Test zur Ausgabe \r\n");
        fw.write("2. Zeile: in eine Datei.");
    } catch (IOException e) {
        System.out.println("Demo failed with: " + e.toString());
    }

    finally {
        if (fw != null) {
            try {
                fw.close();
            } catch (Exception e) {
                // ignore
            }
        }
    }
}
```

## Beispiel FileWriter mit try-with-resources

```

public static void main(String[] args) {

    String fileName = TestData.getAbsoluteFilename("file-writer-demo.txt");

    // try with resources, the writer will be automatically closed at the end
    try (FileWriter writer = new FileWriter(fileName)) {
        writer.write("1. Zeile: Test zur Ausgabe \r\n");
        writer.write("2. Zeile: in eine Datei.");
    } catch (IOException e) {
        System.out.println("Demo failed with: " + e.toString());
    }
}

```

- ▶ Try-with-resources geht für alle Resourcen, die das **Closable** Interface implementieren

## BufferedWriter

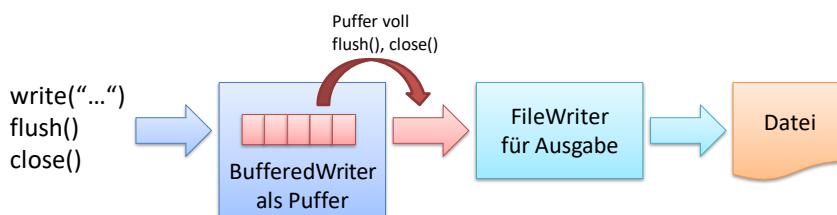
- ▶ Die Ausgabe von einzelnen Zeichen zu einem **Ausgabe** Device ist **zeitaufwändig** (langsam)
- ▶ Effektiver ist es, diese zu puffern und anschliessend **zusammen auszugeben**
- ▶ Für das **puffern** der Daten steht einem dazu der BufferedWriter zur Verfügung
- ▶ Ruft man die write Methode des BufferedWriter auf, so gibt er die Daten nicht aus, sondern speichert diese in einem internen Buffer
  - bis der **Buffer voll** ist
  - oder die **flush()** Methode aufgerufen wird
  - oder der Stream mit **close()** geschlossen wird

## BufferedWriter II

- ▶ Sobald einer der drei genannten Bedingungen eintrifft werden die Daten ausgegeben
- ▶ Die Ausgabe erfolgt aber nicht direkt durch den BufferedWriter, sondern über einen beliebigen Writer des Streaming API
- ▶ Dies hat den Vorteil, dass man den BufferedWriter für eine beliebige Ausgabe einsetzen kann
- ▶ Damit dies funktioniert, wird dem BufferedWriter bei der Erstellung der gewünschte Writer für die Ausgabe mitgegeben

## Beispiel BufferedWriter

- ▶ Im folgenden Beispiel werden Daten mit einem BufferedWriter in eine Datei geschrieben
- ▶ Für die Ausgabe in die Datei wird ein FileWriter verwendet



## Beispiel BufferedWriter II

```

BufferedWriter writer = null;
try {
    writer = new BufferedWriter(new FileWriter("buffered-writer-demo.txt"));

    for (int i = 0; i < 100; i++) {
        writer.write("Buffered Writer Demo");
        writer.newLine();
    }

} catch (IOException e) {
    System.out.println("Demo failed with: " + e.toString());
}

} finally {
    if (writer != null) {
        try {
            writer.close();
        } catch (Exception e) {
            // ignore
        }
    }
}

```

Erstellung BufferedWriter

Übergabe FileWriter  
an BufferedWriter für  
Ausgabe in Datei

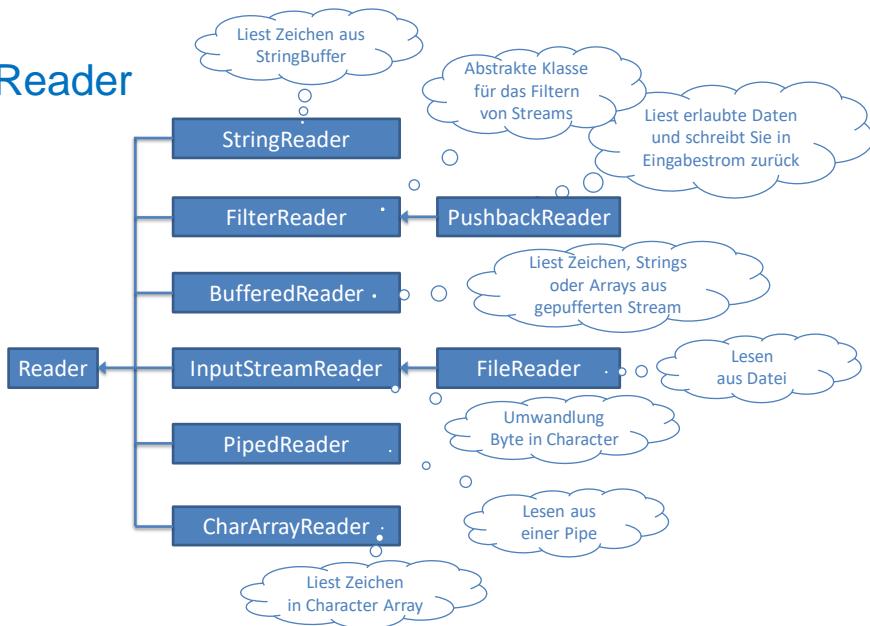
Copyright © iten-engineering.ch

Java Einführung

421

421

## Reader



Copyright © iten-engineering.ch

Java Einführung

422

422

## FileReader

- ▶ Mit dem FileReader werden **Daten** aus einer **Datei** gelesen
- ▶ Wenn bei der Erstellung eines Reader die angegebene Datei nicht vorhanden ist, so wird eine FileNotFoundException geworfen
- ▶ Für das Einlesen stehen anschliessend verschiedene **read** Methoden zur Verfügung
  - Dabei kann man Zeichenweise die Daten einlesen bis zum «end of file»
  - Dies wird mit dem Wert -1 angegeben
- ▶ Am Ende wird der Stream mit **close** geschlossen

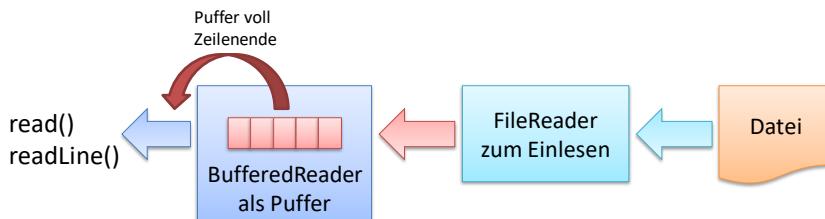
## Beispiel FileReader

```
FileReader reader = null;
try {
    reader = new FileReader("file-reader-demo.txt");
    // read until end of file (-1)
    StringBuilder text = new StringBuilder();
    int x = reader.read();
    while (x != -1) {
        char c = (char) x;
        text.append(c);
        x = reader.read();
    }
    System.out.println(text.toString()); // print text to console
} catch (...) {
} finally {
    ...
}
```

The code demonstrates how to read the content of a file named "file-reader-demo.txt". It uses a `FileReader` object and a `StringBuilder` to build the text. The `try` block handles the file opening and reading. The `finally` block ensures the reader is closed even if an exception occurs.

## BufferedReader

- Möchte man Eingabeströme puffern, so kann man das gleiche Prinzip wie bei den Writer verwenden, nur in umgekehrter Richtung
- Dabei muss dem BufferedReader ein beliebiger Reader für das Lesen der Daten vom gewünschten Device übergeben werden, wie zum Beispiel ein FileReader für das Lesen aus einer Datei



## Beispiel BufferedReader II

```

BufferedReader reader = null;
try {
    // create reader
    reader = new BufferedReader(new FileReader("buffered-reader-demo.txt"));

    // read lines
    String line = reader.readLine();
    while (line != null) {
        System.out.println(line);
        line = reader.readLine();
    }
} catch (IOException e) {
    ...
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (Exception e) {}
    }
}

```

Erstellung BufferedReader

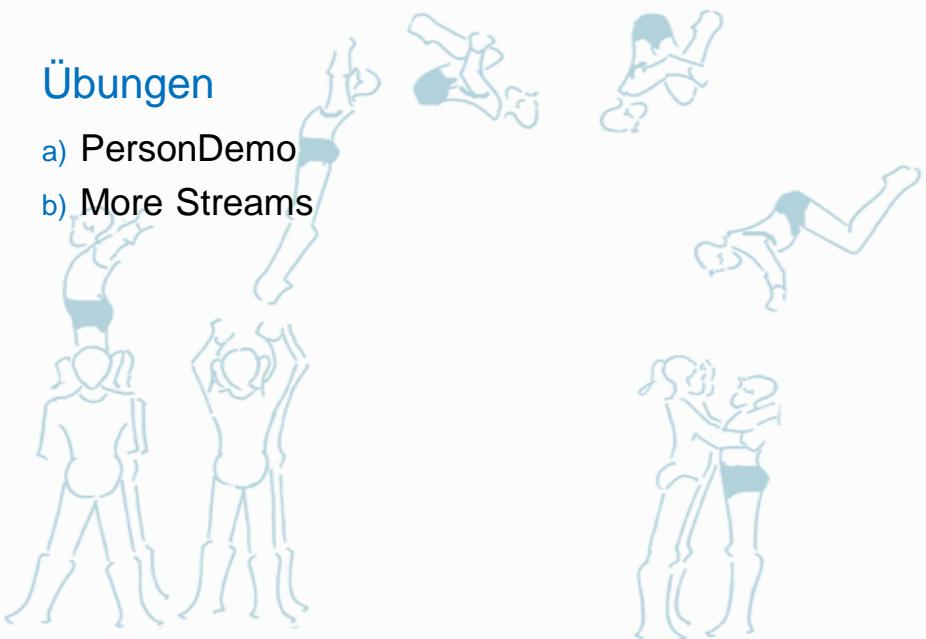
Übergabe FileReader das Lesen der Daten aus der Datei

Daten zeilenweise lesen

## Übungen

a) PersonDemo

b) More Streams



Copyright © iten-engineering.ch

Java Einführung

427

427

## Abschnitt III

Charakter Streams  
mit verschiedenen Datentypen

Copyright © iten-engineering.ch

Java Einführung

428

428

## PrintWriter

- ▶ Die bisherigen Klassen verfügten nur über die `write` Methoden der Superklasse `Writer`
- ▶ Die Klasse `PrintWriter` stellt für jeden **primitiven Datentypen**, für **Strings** und **Objekte** je **eigene Methoden** für die Ausgabe zur Verfügung
  - Mit `print` werden die übergebenen Typen ausgegeben
  - Mit `println` wird zusätzlich noch ein Zeilenumbruch angehängt
  - Für formatierte Ausgabe steht zusätzlich die Methode `printf` zur Verfügung
- ▶ Die Ausgaben von `PrintWriter` werden automatisch **gepuffert**

## PrintWriter erstellen

- ▶ Die Konstruktoren der Klasse `PrintWriter` verlangen als Argument entweder einen `Writer`, ein `OutputStream`, ein `File` oder einen `Dateinamen`
- ▶ Mit dem zusätzlichen Parameter `autoflush` wird festgelegt, wann der interne Buffer geleert werden soll
  - Bei `true` wird der Buffer nach jeden `println` geleert oder wenn er voll ist
  - Bei `false` sobald der Buffer voll ist
- ▶ Zusätzlich gibt es Konstruktoren bei denen der Verwendete **Charakter Set** eingestellt werden kann

## Beispiel PrintWriter

```

public static void main(String[] args) {
    PrintWriter writer = null;
    try {
        writer = new PrintWriter("print-writer-demo.txt");
        writer.println("Ausgabe des Flächeninhalts für Kreise:");
        for (int r = 1; r <= 10; r++) {
            writer.print("Radius r = " + r + ": ");
            writer.println(PI * r * r);
        }
    } catch (IOException e) {
        System.out.println("Demo failed with: " + e.toString());
    } finally {
        // close writer
    }
}

```

## Reader für verschiedene Datentypen

- ▶ Für das Lesen von verschiedenen Datentypen gibt es **keinen Reader** analog dem **PrintWriter**
- ▶ Für das Lesen von Strings kann z.B. der bereits bekannt **BufferedReader** verwendet werden
- ▶ Andere Datentypen müssen zuerst als **String eingelesen** werden und anschliessend im Programm in den gewünschten Typ **konvertiert** werden

## Beispiel

### Reader für verschiedene Datentypen

```

public static String readString() throws IOException {
    BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
    return reader.readLine();
}
public static int readInt() throws NumberFormatException, IOException {
    return Integer.parseInt(readString());
}
public static double readDouble() throws NumberFormatException, IOException{
    return Double.parseDouble(readString());
}
public static void main(String[] args) {
    System.out.println("String = ");
    String s = readString();

    System.out.println("int = ");
    int i = readInt();

    System.out.println("double = ");
    double d = readDouble();
}

```

The code snippet illustrates reading input from the standard input stream (System.in) using a BufferedReader. It defines three methods: readString(), readInt(), and readDouble(). The readString() method reads a line of text. The readInt() and readDouble() methods use Integer.parseInt() and Double.parseDouble() respectively to convert the read string into an integer and a double, respectively. Two callout boxes with arrows point to these conversion lines. The top arrow points to the line 'return Integer.parseInt(readString());' with the text 'Konvertierung nach int'. The bottom arrow points to the line 'return Double.parseDouble(readString());' with the text 'Konvertierung nach double'.

## Abschnitt IV

### Charakter Streams Filter

## FilterWriter & FilterReader

- ▶ Mit Hilfe der beiden abstrakten Klassen FilterWriter und FilterReader ist es möglich die **Daten** während der Verarbeitung **zu filtern**
- ▶ Zu diesem Zweck erstellt man eine **eigene Filter Klasse** die eine der beiden abstrakten Klassen erweitert
  - Im Konstruktor der Klasse wird dabei der Konstruktor der Superklasse aufgerufen
  - Im weiteren sind die **write** bzw. **read Methoden** der Superklasse von der Filterklasse zu **überschrieben**

## BeispielUpperCaseWriter

```
public class UppercaseWriter extends FilterWriter {
    protected UppercaseWriter(Writer writer) {
        super(writer);
    }

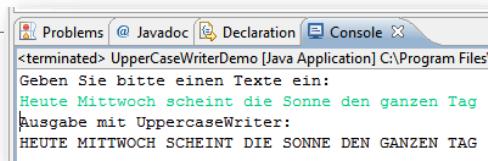
    @Override
    public void write(int c) throws IOException {
        char ch = Character.toUpperCase((char) c);
        super.write((int) ch);
    }

    @Override
    public void write(char[] cbuf, int off, int len) throws IOException {
        for (int i = 0; i < len; i++)
            this.write(cbuf[i + off]);
    }

    @Override
    public void write(String str, int off, int len) throws IOException {
        this.write(str.toCharArray(), off, len);
    }
}
```

## BeispielUpperCaseWriterDemo

```
// Eingabe  
System.out.println("Geben Sie bitte einen Texte ein:");  
  
BufferedReader reader =  
    new BufferedReader(new InputStreamReader(System.in));  
  
String text = reader.readLine();  
  
// Filter und Ausgabe  
PrintWriter writer = new PrintWriter(  
    new UpperCaseWriter(new OutputStreamWriter(System.out)));  
  
System.out.println("Ausgabe mit UppercaseWriter:");  
writer.println(text);  
  
...
```



## Abschnitt V

### Charakter Streams mit Strings und Character Arrays

## StringWriter & CharArrayWriter

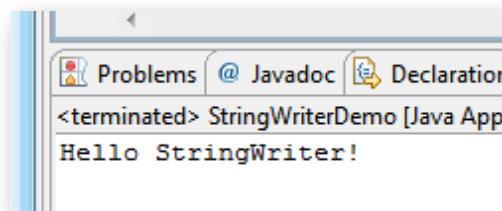
- ▶ Mit dem StringWriter und CharArrayWriter stehen auch Klassen zur Verfügung für die Ausgabe von Daten in String und Character Arrays
- ▶ Wenn Daten ausgegeben werden, wachsen die Puffer automatisch mit
- ▶ Beide Klassen sind von der Klasse Writer abgeleitet und stellen entsprechenden Methoden für die Ausgabe bereit

## Beispiel StringWriter

```
public static void main(String[] args) {
    StringWriter writer = new StringWriter();

    writer.write("Hello");
    writer.append(" ");
    writer.append("String");
    writer.append("Writer");
    writer.append("!\\n");

    System.out.println(writer.toString());
}
```



## StringReader & CharArrayReader

- ▶ Mit dem StringTokenizer und CharArrayReader können auch Daten aus **String** und **Character Arrays** gelesen werden
- ▶ Beide Klassen sind von der Klasse Reader abgeleitet und stellen entsprechenden Methoden für das Lesen bereit
- ▶ Mit Hilfe der beiden Klassen StringTokenizer und StringTokenizer ist es somit möglich, **Strings** auf die **gleiche Art und Weise** zu verarbeiten wie **andere Daten Streams**

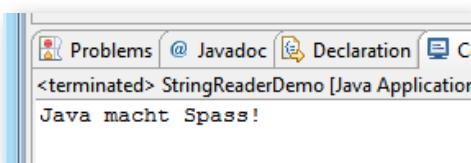
## Beispiel StringTokenizer

```
public static void main(String[] args) {
    final int EOF = -1;
    String string = "Java macht Spass!";

    StringTokenizer reader = new StringTokenizer(string);

    try {
        int c = reader.read();

        while (c != EOF) {
            System.out.print((char) c);
            c = reader.read();
        }
    } catch (IOException io) {
        System.out.println(io.getMessage());
    }
}
```



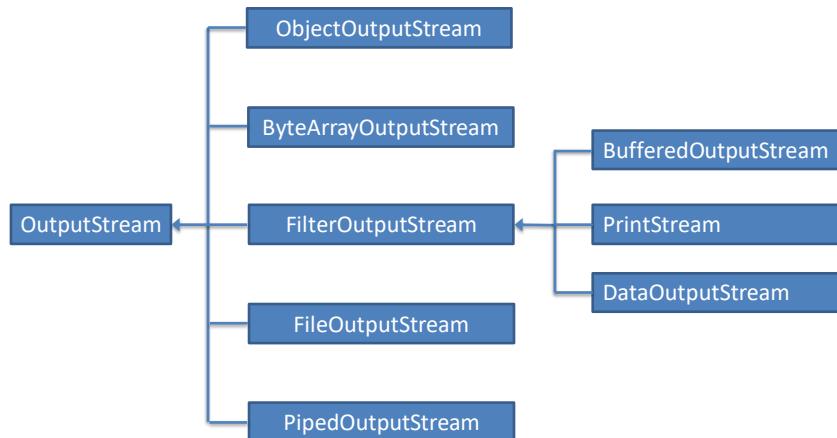
## Abschnitt VI

### Byte Streams

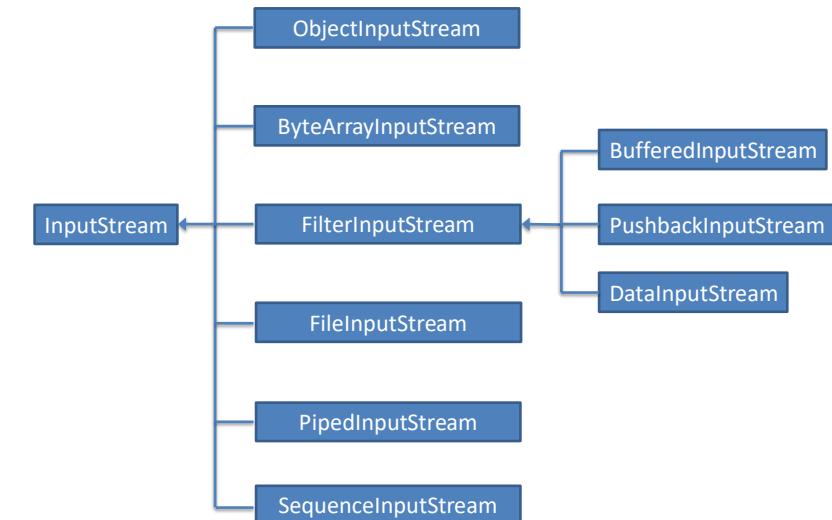
### Byte Streams

- ▶ Byte Streams transportieren jeweils 8 Bit
- ▶ Sie stellen die grundlegende Ein- und Ausgabe von Byte Daten dar
- ▶ Alle Klasse für die Ausgabe von Byte Streams werden von der Superklasse **OutputStream** abgeleitet
- ▶ Alle Klasse für die Eingabe von Byte Streams werden von der Superklasse **InputStream** abgeleitet

## OutputStream



## InputStream



## Byte- und Character Streams

- ▶ Viele Byte Stream Klassen haben ein Pendant bei den Character Stream Klassen
- ▶ Sie verfügen meist über die gleichnamige Methoden und unterscheiden sich nur durch die Verarbeitungsbreite
- ▶ Als Schnittstelle zwischen den beiden Stream Typen dienen die Klassen `InputStreamReader` und `OutputStreamWriter`

## Standard Ein- und Ausgabe

- ▶ Das Handling für die Standard Ein- und Ausgabegeräte (meist Tastatur und Bildschirm) ist in der Klasse `java.lang.System` implementiert
- ▶ System verwendet dabei die beiden Klassen `InputStream` und `PrintStream` aus dem Stream Package
- ▶ In den beiden Stream Klassen sind `read` und `print` Methoden definiert, die für die Ein- und Ausgabe auf die Standardgeräte genutzt werden können

## Standard Ein- und Ausgabe II

| Standard            | Konstanten der Klasse System               | Aufruf im Code |
|---------------------|--------------------------------------------|----------------|
| Eingabestrom        | public static final InputStream <b>in</b>  | System.in      |
| Ausgabestrom        | public static final PrintStream <b>out</b> | System.out     |
| Fehler-ausgabestrom | public static final PrintStream <b>err</b> | System.err     |

```
public static void main(String[] args) {
    byte[] b = new byte[1];

    try {
        System.out.print("Bitte geben Sie ein Zeichen ein: ");
        System.in.read(b);
        System.out.println((char) b[0] + " hat den ASCII-Code " + b[0]);
    } catch (IOException io) {
        System.out.println(io.getMessage());
    }
}
```

## Kapitel 18

Random, Date & Time,  
System, Console

## Random

- ▶ Für die Generation von Zufallszahlen gibt es in Java zwei Varianten
- ▶ Die Methode `random` der Klasse `java.lang.Math`
  - Die Methode liefert eine Zufahlszahl vom Typ `double` zwischen 0.0 und 1.0
- ▶ Oder die Klasse `java.util.Random`
  - Bietet verschiedene Methoden zur Generierung von Zufallszahlen

```
Random random = new Random();
int number;

// Liefert eine Zufallszahl zwischen 0..99
number = random.nextInt(100)
```

## Date & Time

- ▶ Seit Java Version 1 steht die Klasse `Date` zur Darstellung und Berechnung von Datums- und Zeitwerten im JDK zur Verfügung
  - Mittlerweile sind viele Methoden von Date als `deprecated` markiert
  - Dass hiesst, diese Methoden sollte man nicht mehr einsetzen
- ▶ Ab Java 1.1 wurde stattdessen die abstrakte Klasse `Calendar` und davon abgeleitet die Klasse `GregorianCalendar` angeboten
  - Damit lassen sich umfangreiche Datums- und Zeitberechnungen machen

## Neue java.time API

- ▶ Seit Java 8 steht im Package [java.time](#) eine neue API zur Verfügung
- ▶ Diese ist [vergleichbar](#) mit der populären [Joda Time Library](#) (ist aber nicht dasselbe)
- ▶ Die neue API bietet eine Vielzahl von neuen Klassen für die Berechnung von Datumswerten und Zeitangaben
- ▶ Falls man also bereits Java 8 im Einsatz hat, sollte man ungedingt von den neuen Möglichkeiten profitieren

## Merkmale

- ▶ Unterstützung vom [ISO 8601](#) Zeitstandard
- ▶ [Internationale Kalender](#) wie z.B. der japanische, islamische (Hijriah), buddhistische oder in der in China häufig verwendete Minguo werden unterstützt
- ▶ Die API kann mit weiteren Klassen (zum Beispiel für spezielle Kalender) ergänzt werden
- ▶ Alle `java.time` Objekte sind [immutable](#)

## Regeln

- ▶ Ein Tag hat genau 86'400 Sekunden
  - Schaltsekunden werden nicht berücksichtigt
- ▶ Die Genauigkeit der Berechnung erfolgt in Nanosekunden
- ▶ Um Mitternacht entspricht die Zeit genau der öffentlichen Zeit
- ▶ Zu allen anderen Zeitpunkten entspricht die Java Zeit so genau wie möglich der offiziellen Zeit
  - Zur Berechnung wird ein exakt definierter Algorithmus verwendet

## Beispiele

- ▶ Das package `java.time` umfasst u.a. die folgenden Klassen

| Klassen                      | Einsatz                                                                           |
|------------------------------|-----------------------------------------------------------------------------------|
| Instant, Duration            | Die Berechnung von Zeitpunkten und deren Differenz                                |
| LocalDate, Period            | Das Arbeiten mit Datumsangaben und Perioden                                       |
| LocalDateTime, ZonedDateTime | Das Arbeiten mit Datumsangaben mit und ohne Zeitzonen                             |
| TemporalAdjuster             | Kalendermanipulationen (wie zum Beispiel das bestimmen des ersten Montag im Jahr) |
| LocalTime                    | Berechnung von Zeiten                                                             |
| DateTimeFormatter            | Formatierung und Darstellung von Datum und Zeit                                   |
| Clock                        | Aktuelles Datum mit Zeit und Zeitzone                                             |

## System

- ▶ Die Klasse `java.lang.System` bietet u.a. Methoden für folgende Bereich an:
- ▶ Ausgabe von **Systeminformationen**
- ▶ Exit Methode für die Beendigung einer Anwendung und Rückgabe eines Status Werts
- ▶ Definition der **Standard Ein- und Ausgabe Streams** sowie Methoden für die Umleitung dieser Streams
- ▶ Angabe der **Systemzeit**
- ▶ Aufruf **Garbage Collektor** (nicht zwingend)

## Console

- ▶ Mit Java SE 6 wurde die Klasse `java.io.Console` eingeführt
- ▶ Diese ist für die Ein- und Ausgabe von Textinformationen über die Konsole vorgesehen
- ▶ Unter anderem ist es nun auch möglich, ein Passwort ohne Anzeige einzulesen
- ▶ Mit der Methode `system.console` wird ein neues Objekt erzeugt, oder null zurückgegeben falls die Anwendung keine Console hat

## Console II

### Beachte

Whether a virtual machine has a console is dependent upon the underlying platform and also upon the manner in which the virtual machine is invoked.

If the virtual machine is started from an interactive command line without redirecting the standard input and output streams then its console will exist and will typically be connected to the keyboard and display from which the virtual machine was launched.

**If the virtual machine is started automatically, for example by a background job scheduler, then it will typically not have a console.**

Quelle: Javadoc Console ( <http://java.sun.com/javase/6/docs/api> )

Copyright © iten-engineering.ch

Java Einführung

459

459

## Beispiel Console

```
public static void main(String[] args) {
    Console console = System.console();

    // Prüfung ob Console Objekt verfügbar
    if (console == null) {
        System.exit(1);
    }

    // Eingabe und Passwort Check
    char[] pw = console.readPassword("Bitte geben Sie das Passwort ein:");

    // compare
    if (Arrays.equals(pw, SECRET_PW)) {

        // Passwort löschen
        Arrays.fill(pw, 'x');

        // Start Verarbeitung
        ...

    } else {
        // Abbruch
        System.exit(-1);
    }
}
```

Passwort von der Konsole einlesen

Copyright © iten-engineering.ch

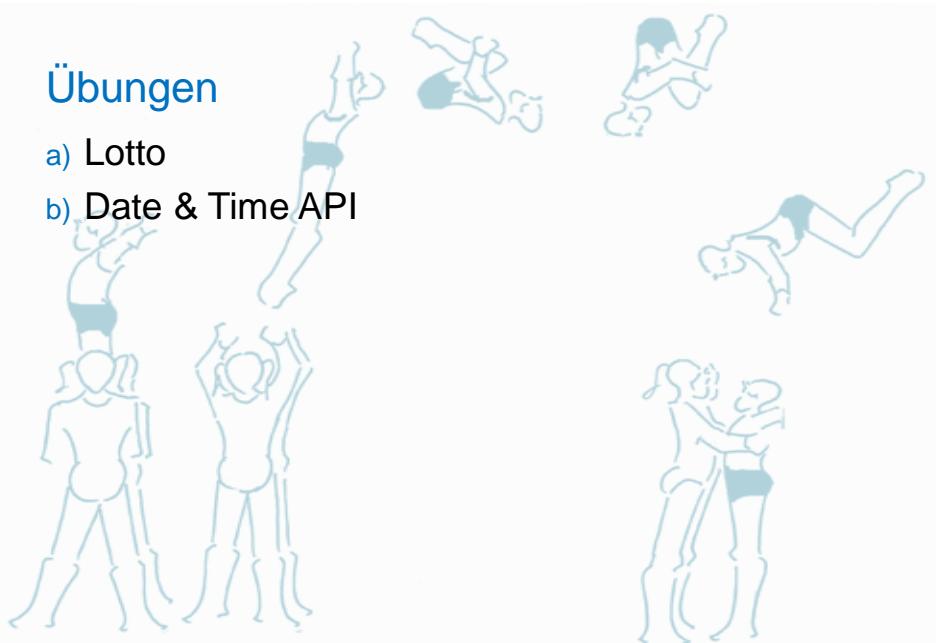
Java Einführung

460

460

## Übungen

- a) Lotto
- b) Date & Time API



Copyright © iten-engineering.ch

Java Einführung

461

461

## Kapitel 19

### Funktionale Programmierung

Copyright © iten-engineering.ch

Java Einführung

462

462

## Funktionale Programmierung

- ▶ Funktionale Programmiersprachen gibt es mit Lisp und Scheme schon lange
- ▶ Mit **Scala** und Clojure (beide erzeugen Byte Code der auf der JVM ausgeführt wird) hat die funktionale Programmierung in letzter Zeit wieder an **Bedeutung** gewonnen
- ▶ Mit den **Lambda Expression** bietet nun auch **Java** seit Version **8** Möglichkeiten der funktionalen Programmierung an!

## Lambda Expression

- ▶ Ein Lambda Ausdruck wird im Rahmen der funktionalen Programmierung als **anonyme Funktion** bezeichnet
- ▶ Es ist ein Block **Programmcode** mit Parametern, der im Programm „herumgereicht“
- ▶ Der Code kann dann zu einem späteren Zeitpunkt ein- oder mehrfach **ausgeführt** werden

## Demo Runnable

```
public static void main(String[] args) {

    // Anonyme Klasse die das Interface Runnable implementiert.
    // - das Interface hat die Methode: void run();
    // - die run Methode hat keine Parameter und kein Returnwert
    Runnable r1 = new Runnable() {
        @Override
        public void run() {
            System.out.println("Hello world.");
        }
    };

    r1.run();

    // Lambda Ausdruck anstelle der anonymen Klasse.
    // - der Ausdruck hat keine Parameter und gibt keinen Wert zurück.
    Runnable r2 = () -> System.out.println("Hello Lambda world!");

    r2.run();
}
```

Copyright © iten-engineering.ch

Java Einführung

465

465

## Funktionale Interfaces

- ▶ Die Übergabe des Codes erfolgt an **funktionale Interfaces**
- ▶ Diese definieren genau **eine abstrakte Methode**
- ▶ Vor Java 8 wurden für solche Situationen anonyme Klassen genutzt, welche die Methode eines Interfaces implementierten
- ▶ Lambda Ausdrücke können im Gegensatz zu einer anonymen Klasse viel **kompakter** dargestellt werden
- ▶ Der Code wird dadurch **einfacher** und **übersichtlicher**

Copyright © iten-engineering.ch

Java Einführung

466

466

## Demo ActionListener

```
// Anonyme Klasse die das Interface ActionListener implementiert.
// - das Interface hat die Methode:
//   void actionPerformed(ActionEvent event);
// - die Methode hat einen Parameter und keinen Returnwert.
 JButton btn1 = new JButton("Button 1");
 btn1.addActionListener(new ActionListener() {
     @Override
     public void actionPerformed(ActionEvent ae) {
         System.out.println("Click detected by anonymous class.");
     }
 });

// Lambda Ausdruck anstelle der anonymen Klasse.
// - der Ausdruck hat einen Parameter und gibt keinen Wert zurück.
 JButton btn2 = new JButton("Button 2");
 btn2.addActionListener(
     e -> System.out.println("Click detected by lambda listner."))
 );
```

## Syntax

Beispiel: **(int x, int y) -> { return x + y; }**

- ▶ Die **Parameterliste** enthält die Parameter der Methode des funktionalen Interfaces
  - Die Typenangaben sind optional, falls der Compiler diese aus dem Kontext ableiten kann
  - Die Klammern sind optional, falls nur ein Parameter übergeben wird (Beachte: falls es keine Parameter hat, sind die Klammern anzugeben)
- ▶ Der **Pfeil Operator** steht zwischen Parameter und Anweisungscode
- ▶ Der **Anweisungscode** kann aus 1..n Anweisungen mit oder ohne Return Wert bestehen
  - Die Klammern {} sind optional, falls es nur eine Anweisung hat
  - Bei einem Rückgabewert ist die Angabe von return optional, falls es keine {} hat
  - Die Schlüsselwörter break und continue sind nur innerhalb von Schleifen erlaubt
- ▶ Der Zugriff auf final Variablen des umschliessenden Bereichs ist erlaubt

## Demo PersonComparator

```

List<Person> personList = new ArrayList<Person>();
personList.add(new Person("Pipi", "Langstrumpf"));
personList.add(new Person("Michel", "von Löneberga"));
personList.add(new Person("Peter", "Pan"));

// Anonyme Klasse die das Interface Comparator implementiert.
// - das Interface hat die Methode: int compare(T o1, o2);
// - die compare Methode hat zwei Parameter und gibt einen
//   int Wert mit dem Resultat des Vergleichs zurück.
Collections.sort( personList, new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
        return p1.getFirstname().compareTo(p2.getFirstname());
    }
});
```

## Demo PersonComparator II

```

// Der Lambda Ausdruck hat 2 Parameter vom Typ Person und gibt einen
// Returnwert vom Typ int zurück.
Collections.sort(personList, (Person p1, Person p2) ->
    { return p1.getFirstname().compareTo(p2.getFirstname()); });

// Mit dem zweiten Lambda Ausdruck wird die Sortierung umgekehrt.
// Im weiteren ist folgendes zu beachten:
// 1) Bei den Parametern wird der Typ nicht angegeben.
//    Dieser ist optional falls der Compiler dies aus dem Context
//    herausfinden kann (target typing).
// 2) Die Anweisung verwendet keine Klammern {}.
//    Diese sind optional falls es nur eine Anweisung hat
// 3) Die Anweisung verwendet kein return statement.
//    Dies ist optional, falls keine Klammern {} verwendet werden.
Collections.sort(personList, (p1, p2) ->
    p2.getFirstname().compareTo(p1.getFirstname()));
```

## Einsatz

- ▶ Der Einsatz von Lambda Ausdrücken ist überall dort möglich, wo ein functional Interface erwartet wird
- ▶ Dies kann mit der entsprechenden Annotation `@FunctionalInterface` gekennzeichnet werden
- ▶ Mit Java 8 wurden diverse APIs entsprechend angepasst, so zum Beispiel bei der nebenläufigen Programmierung oder der Verarbeitung von Dateien
- ▶ Zudem wurde das Collection Framework mit einer neuen Stream API erweitert, bei dem parallele Zugriffe auf Collections möglich sind

## @FunctionalInterface

- ▶ Damit die Lambda Ausdrücke mit Java 8 eingeführt werden können, wurden folgende Spracherweiterungen eingebaut:
  - Referenzen auf Methoden
  - Interface mit Default Methoden
  - Funktionale Interface
- ▶ Ein functional Interface hat neben beliebig vielen default und statischen Methoden genau eine abstrakte Methode

## @FunctionalInterface II

```
@FunctionalInterface
public interface Runnable {

    void run();
}
```

- ▶ Falls ein Interface mit @FunctionalInterface markiert wird und mehrere abstrakte Methoden hat so gibt es einen Compile Fehler.
- ▶ Beachte:
  - Für die Ausführung von Lambda Ausdrücken ist die @FunctionalInterface Annotation **nicht zwingend**.
  - Voraussetzung ist einzig ein Interface mit **einer abstrakten Methode**

## Demo Converter

```
@FunctionalInterface
interface Converter {
    double convert(double input);
}

private static void calculate(double[] values, Converter converter) {
    for (double value : values) {
        System.out.println(value + " = " + converter.convert(value));
    }
}

public static void main(String[] args) {
    // Temperaturen in Grad Celsius
    double[] celsius = { 18, 19, 20, 21, 22, 23 };

    // K = C + 273.15
    System.out.println("Grad Celsius in Kelvin:");
    calculate(celsius, input -> input + 273.15);

    // F = C x 1.8 + 32
    System.out.println("Grad Celsius in Fahrenheit:");
    calculate(celsius, input -> (input * 1.8) + 32);
}
```

```
Grad Celsius in Kelvin:
18.0 = 291.15
19.0 = 292.15
20.0 = 293.15
21.0 = 294.15
22.0 = 295.15
23.0 = 296.15
Grad Celsius in Fahrenheit:
18.0 = 64.4
19.0 = 66.2
20.0 = 68.0
21.0 = 69.80000000000001
22.0 = 71.6
23.0 = 73.4
```

## Vordefinierte Interface

- Im package `java.util.function` bietet Java bereits eine Reihe von vordefinierten funktionalen Interface, unter anderem die folgenden:

| Interface                                                                                                                   | Methode                                                                                                             |
|-----------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| <code>Consumer&lt;T&gt;</code><br>T - the type of the input to the function                                                 | <code>void accept(T t)</code><br>Represents an operation that accepts a single input argument and returns no result |
| <code>Function&lt;T,R&gt;</code><br>T - the type of the input to the function<br>R - the type of the result of the function | <code>R apply(T t)</code><br>Represents a function that accepts one argument and produces a result.                 |
| <code>Predicate&lt;T&gt;</code><br>T - the type of the input to the function                                                | <code>boolean test(T t)</code><br>Represents a predicate (boolean-valued function) of one argument.                 |
| <code>Supplier&lt;T&gt;</code><br>T - the type of results supplied by this supplier                                         | <code>T get()</code><br>Represents a supplier of results.                                                           |

## Demo Converter2

```
public class ConverterDemo2 {

    private static void calculate(double[] values, Function<Double, Double> converter) {
        for (double value : values) {
            System.out.println(value + " = " + converter.apply(value));
        }
    }

    public static void main(String[] args) {
        // Temperaturen in Grad Celsius
        double[] celsius = { 18, 19, 20, 21, 22, 23 };

        // K = C + 273.15
        System.out.println("Grad Celsius in Kelvin:");
        calculate(celsius, input -> input + 273.15);

        // F = C * 1.8 + 32
        System.out.println("Grad Celsius in Fahrenheit:");
        calculate(celsius, input -> (input * 1.8) + 32);
    }
} // end
```

```
Grad Celsius in Kelvin:
18.0 = 291.15
19.0 = 292.15
20.0 = 293.15
21.0 = 294.15
22.0 = 295.15
23.0 = 296.15
Grad Celsius in Fahrenheit:
18.0 = 64.4
19.0 = 66.2
20.0 = 68.0
21.0 = 69.80000000000001
22.0 = 71.6
23.0 = 73.4
```

## Methoden Referenzen

- ▶ Neben den Lambda-Ausdrücken gibt es die **Methoden-** und **Konstruktor-Referenzen**, die von der Syntax her noch kompakter als die Lambda-Ausdrücke sind.
- ▶ Wenn man in einem Lambda-Body ohnehin nichts weiter tut, als eine bestimmte Methode aufzurufen, dann kann man den Lambda-Ausdruck häufig durch eine **Methoden-Referenz ersetzen**.
- ▶ Vorteile
  - Kompakter und übersichtlicher
  - Bestehende Funktionalität kann wiederverwendet werden

## 4 Typen von Methoden Referenzen

- ▶ Statische Methode
  - ClassName::methodName
  - Beispiel: String::valueOf
- ▶ Instanzmethode eines Objektes
  - Expr::methodName
  - Beispiel: System.out.println
- ▶ Instanzmethode einer bel. Instanz einer Klasse
  - ClassName::methodName
  - Beispiel: String::length
- ▶ Konstruktor:
  - ClassName::new
  - Beispiel: String::new

## Demo MethodeRef

```
public class MethodeRefDemo {

    public static void main(String[] args) {
        String s = "Ausgabe mit Methodenreferenz bzw. Lambda-Ausdruck";

        // Lambda Ausdruck
        print(() -> s.toString());

        // Methode Referenz
        print(s::toString);
    }

    public static void print(Supplier<String> supplier) {
        System.out.println(supplier.get());
    }
}
```

Copyright © iten-engineering.ch

Java Einführung

479

479

## Demo MethodeRef2

```
public class MethodeRefDemo2 {

    public static void main(String[] args) {

        String[] words = { "3 ", " 2 ", "", " 1" };

        System.out.println("Resultat:");

        Arrays.stream(words)
            .map(String::trim).filter((s) -> !s.isEmpty())
            .map(Integer::parseInt)
            .sorted()
            .forEach(System.out::println);
    }

} // end
```

Resultat:  
1  
2  
3

Copyright © iten-engineering.ch

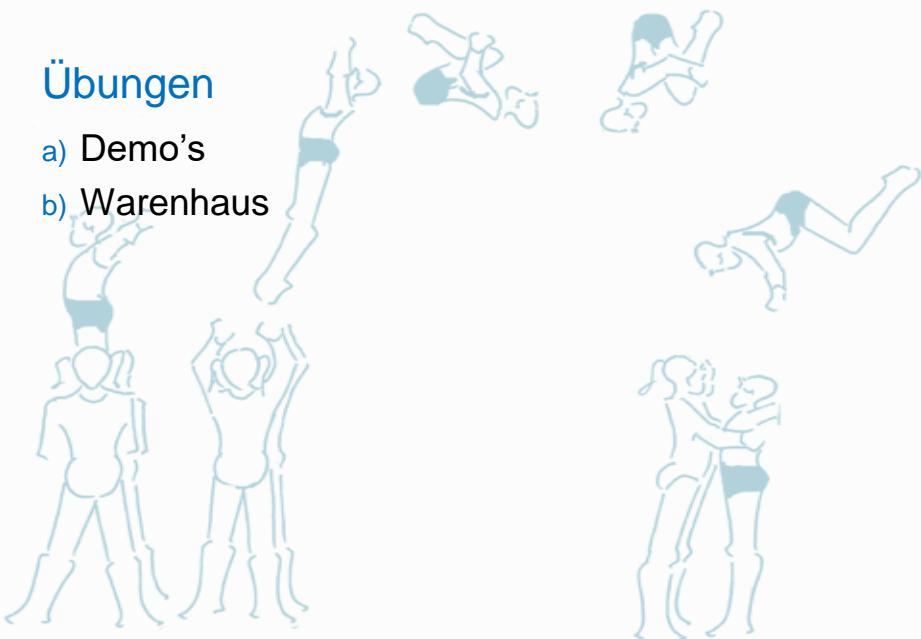
Java Einführung

480

480

## Übungen

- a) Demo's
- b) Warenhaus



Copyright © iten-engineering.ch

Java Einführung

481

481

## Anhang

### Literatur und Weblinks



Copyright © iten-engineering.ch

Java Einführung

482

482

## Literatur und Weblinks

- ▶ Java SE at a Glance  
<http://www.oracle.com/technetwork/java/javase/overview/index.html>
- ▶ Java SE Documentation  
<http://www.oracle.com/technetwork/java/javase/documentation/index.html>
- ▶ Java Tutorials  
<http://docs.oracle.com/javase/tutorial/>
- ▶ Eclipse  
<http://www.eclipse.org/>
- ▶ Eclipse Download  
<http://www.eclipse.org/downloads/>