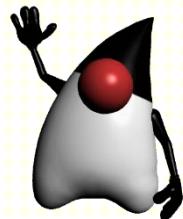


# ***Java Programmierung***



OpenJDK

Michaela Pum



# *Java ist ...*

## ▪ ... Eine Programmiersprache

- Plattformunabhängig
- Objektorientiert
- Fokus liegt auf Kapselung von Daten und Methoden
- Syntaktisch orientiert an C/C++

```
Scanner scanner = new Scanner(System.in);

int l, b;
System.out.println("Bitte die Länge eingeben:");
l = scanner.nextInt();
System.out.println("Bitte die Breite eingeben:");
b = scanner.nextInt();
scanner.nextLine();

int flaeche = l * b;
System.out.printf("Die Fläche beträgt %d \n", flaeche);
scanner.close();
```

## ▪ ... Ein Framework

- Integriert mit umfangreichen Klassenbibliotheken
- Zur Ausführung von Java-Programmen auf unterschiedlichen Plattformen

# Java Frameworks

- **Java SE:**
  - Java Standard Edition (früher J2SE), für Desktop-Anwendungen
- **Java EE:**
  - Java Enterprise Edition (früher J2EE), Spezifikation für verteilte Anwendungen in der Java Plattform
- **Java ME:**
  - Java Micro Edition (früher J2ME), für Anwendungen auf Kleingeräten
- **Java Card:**
  - API für Smartcards
- **Java FX:**
  - Für Rich Client Applications (GUI Anwendungen)
  - Ist mittlerweile Open Source Projekt

- **Java EE**

- Spezifikation für verteilte Anwendungen
- Ausgerichtet auf Mehrschicht-Architekturen
- Unterstützt viele Möglichkeiten der Applikationsverteilung mit Hilfe von unterschiedlichen Java EE Komponenten, z.B.
  - Servlet, JSP, JSF, EJB, REST Service, XML Webservice
- Erfordert zur Ausführung einen Java EE Application Server (z.B. JBoss AS oder Glassfish)

# Geschichte

Jahr	Version	Anmerkung
1992	Vorläufer OAK	Portable Plattform für Videorecorder, Stereoanlagen, Mikrowellen, Sicherheitssysteme, Set-Top-Boxen
1996	Java 1.0	für nichtkommerzielle Zwecke frei
2004	Java 5 (=1.5)	Nachfolger von 1.4, wurde auch als „Java 2 JDK 5“ bezeichnet
2006	Java 6 (=1.6)	seither unter GPL2 verwendbar
2014	Java 8 (=1.8)	Neue Sprachfeatures (Lambda Expressions und Stream API); erste LTS Release, Support bis 2030
2017	Java 9	Open Source Referenzimplementierung

# Geschichte

Jahr	Version	Anmerkung
2017	Java 9	Open Source Referenzimplementierung
2018	Java 11	LTS Release, Support bis 2023, Extended Support bis 2026
2021	Java 17	aktuelle LTS Release, Support bis 2026, Extended Support bis 2029

- **Seit Java 11**

- Neuer Release-Zyklus
  - Etwa alle 6 Monate
    - **STS Release**, Support endet mit nächster Release
  - Etwa alle 3 Jahre
    - **LTS Release**, mit Support für mehrere Jahre

## ■ Seit Java 11

- Release in 2 Versionen



- Oracle JDK

- <https://www.oracle.com/java/technologies/downloads/>
    - kostenpflichtig

- Open JDK

- <https://openjdk.java.net/>
    - open source, gratis verwendbar
    - keine Installationspakete, manuelle Konfiguration



- Dokumentation für beide Versionen

- <https://docs.oracle.com/en/java/javase/17/docs/api/index.html>

- Adoptium OpenJDK

- <https://adoptium.net/>
  - stellt Installationspakete und regelmäßige Updates für Open JDK bereit

# Entwicklungsumgebungen

## ■ Eclipse

- Weit verbreitete Open Source IDE
- Anpassbar durch sehr viele Plugins
- Frei verwendbar



## ■ NetBeans

- Die zu Java gehörende IDE
- Frei verwendbar
- Derzeit von Apache entwickelt



## ■ IntelliJ Idea

- Relativ neue IDE der Firma JetBrains
- für kommerzielle Zwecke kostenpflichtig



# Installation

- **Mit Admin-Rechten**
  - Installationspakte für JDK und Entwicklungsumgebung herunterladen und installieren
  - Umgebungsvariable PATH wird automatisch angepasst
- **Ohne Admin-Rechte**
  - ZIP-Pakete für JDK und Entwicklungsumgebung herunterladen und entpacken
  - in der Umgebungsvariable PATH das bin-Verzeichnis der JDK manuell hinzufügen
    - Unter Windows kann das Verzeichnis in der PATH-Umgebungsvariable des Benutzers hinzugefügt werden, es dürfen aber keine JDKs installiert sein oder bereits in der System-Umgebungsvariable PATH vorkommen

# Projektstruktur

Zu einem Projekt gehören meist mehrere Klassen die in einem eigenen Unterverzeichnis vom CLASSPATH liegen sollten

Klassen die mit dem Interpreter gestartet werden, benötigen eine main-Methode

Klassendefinition

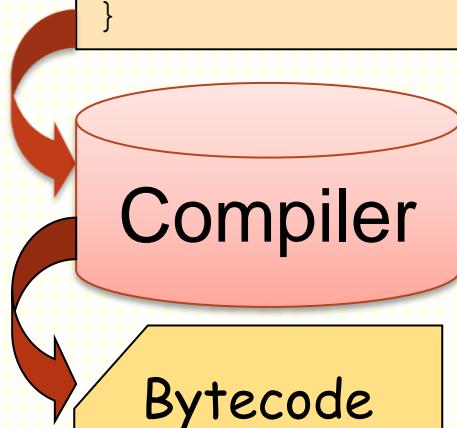
```
package hello.program;

public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello Java World!");
    }
}
```

# Kompilierung

hello/program/HelloWorld.java

```
package hello.program;
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello Java World!");
    }
}
```



Bytecode

....

....

main

... ↓

.. ↓

Übersetzung erfolgt in einen einheitlich genormten Byte-Code

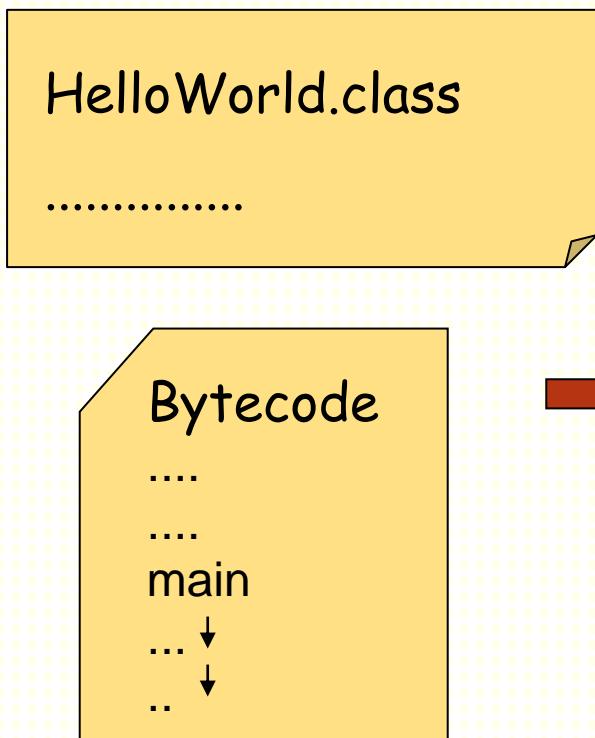
HelloWorld.class

.....

hello/program/HelloWorld.class

# Ausführung

Interpretation des Byte-Code erfolgt durch das Java Runtime Environment (JRE)



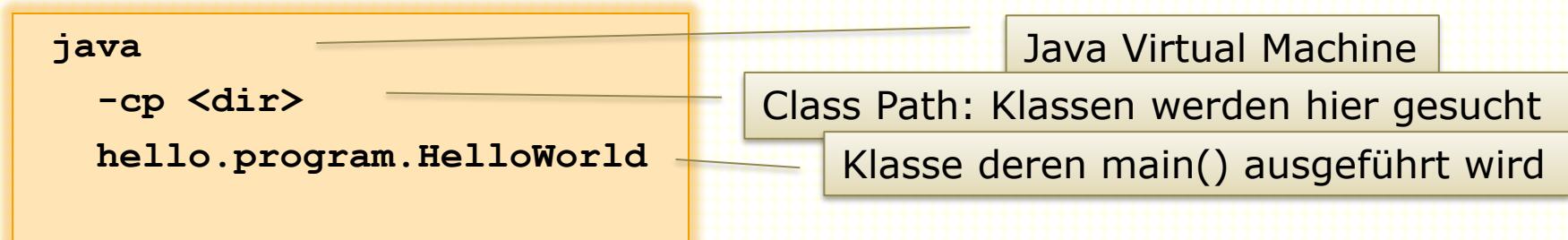
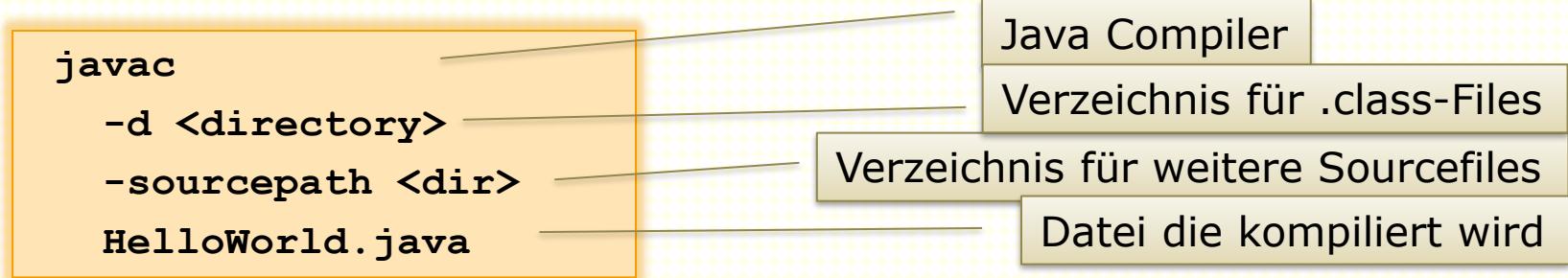
C:\Test>java hello.program.Helloworld  
Hello Java world!

# **Java: JRE und JDK**

- **Java Runtime Environment (JRE)**
  - für die Ausführung von Java Programmen
- **Java Development Kit (JDK)**
  - Programme und Tools für die Entwicklung
  - wurde zeitweise auch Java SDK genannt
- **Bis Java 8**
  - waren JRE und JDK separat
- **Seit Java 9**
  - gibt es nur noch ein Paket

# Kompilierung & Ausführung in der Konsole

## ▪ Java Compiler und Java Runtime Environment



**Java Virtual Machine (JVM, VM) = Java Runtime Environment (JRE)**

```
C:\Test>javac -d bin -sourcepath src src\hello\program\Helloworld.java  
C:\Test>java -cp bin hello.program.Helloworld
```

# ***Java***

Die Programmiersprache

# Namenskonventionen

- **Klassen**
  - Beginnen mit Großbuchstaben (klein weiter)  
Beispiel: class MouseHandler;
- **Identifier für Packages, Variablen und Methoden**
  - Beginnen mit einem Kleinbuchstaben  
Beispiel: int jahreszahl = 1602;
- **Konstante**
  - Bestehen nur aus Großbuchstaben und \_  
Beispiel: final int SHAKE\_SPEAR = 46;

# Primitive Datentypen

Typ	Inhalt	Größe
<b>boolean</b>	Wahrheitswert (true / false) Benötigte Speichergröße: VM-abhängig	1 Bit
<b>char</b>	ein Unicode-Zeichen, unsigned: 0 bis 65535	16 Bit
<b>byte</b>	Ganzzahl, signed -128 bis 127	8 Bit
<b>short</b>	Ganzzahl, signed -32768 bis 32767	16 Bit
<b>int</b>	Ganzzahl, signed $-2^{31}$ bis $2^{31}-1$	32 Bit
<b>long</b>	Ganzzahl, signed $-2^{63}$ bis $2^{63}-1$	64 Bit
<b>float</b>	Fließkommazahl, single-precision IEEE 754 Sign(1) + Exponent(8) + Fraction(23)	32 Bit
<b>double</b>	Fließkommazahl, double-precision IEEE 754 Sign(1) + Exponent(11) + Fraction(52)	64 Bit

Standardtyp für Ganzzahl ist int, für Fließkomma double

# Schreibweise für Literale

Typ	Suffix	Beispiel-Literale
int		5678, 100_000, 0x03B1
long	l, L	5678L, 0x03B1L, 10_000_000_000L
float	f, F	1.234F
double	d, D	234.78, 234.78d
char		'x', '1'
String		"x", "1", "Hallo!", ""

Literal = Hartcodierter konstanter Wert im Sourcecode

```
int i = 5678;          // 5678      hat Typ int
long l = 5678L;        // 5678L     hat Typ long
float f = 1.234F;       // 1.234F    hat Typ float
double d = 234.78;      // 234.78    hat Typ double
char c = '1';           // '1'       hat Typ char
String s = "Hallo";    // "Hallo"   hat Typ String
```

# Schreibweise für Literale

- **Escape-Sequenzen für Sonderzeichen**
  - für char und einzelne Zeichen eines Strings
  - werden mit Backslash (\) eingeleitet

	Bedeutung
\n	Zeilenvorschub (ASCII 10)
\r	Wagenrücklauf (ASCII 13)
\t	Tabulator
'	einfaches Hochkomma
"	doppeltes Hochkomma
\\"	Backslash
\unnnn	das Zeichen mit dem Unicode-Wert nnnn

```
System.out.println('\\u03B1'); // griech. α  
System.out.println("C:\\Java\\Programme");  
System.out.println("Hallo \"Java\"");
```

α  
C:\Java\Programme  
Hallo "Java"

# Schreibweise für Literale

## ■ Textblöcke

- mehrzeiliges String-Literal (seit Java 14)

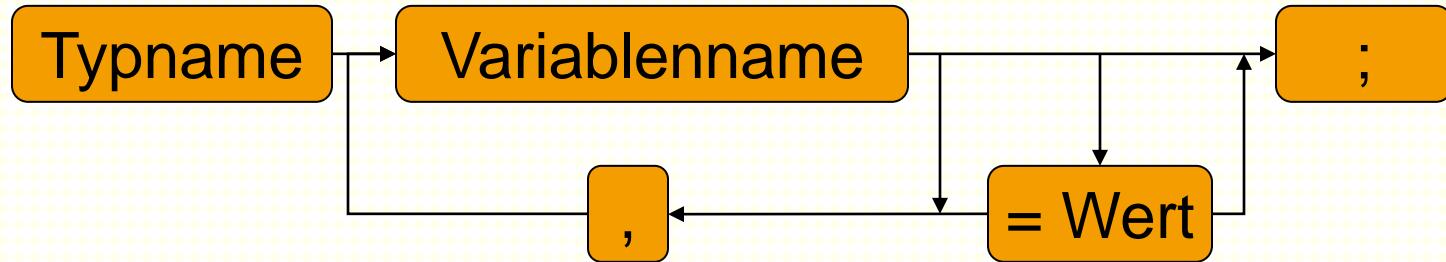
	Bedeutung
"""	Beginn und Ende des Textblocks
" (einzeln)	ist ein normales Zeichen
\	fügt den Text der folgenden Zeile ohne Umbruch hinzu
\s	Leerzeichen, das nicht abgeschnitten wird

```
String info = """  
Name: "Elefant"  
Region:  
Afrika, südlich der Sahara, \  
Asien, Indien, Srilanka und Sundainseln""";  
System.out.println(info);
```

```
Name: "Elefant"  
Region:  
Afrika, südlich der Sahara, Asien, Indien, Srilanka und Sundainseln
```

# Deklaration von Variablen

- Deklaration



```
public static void main(String args[]) {  
    int a = 46;  
    int b, c;  
    double d, e = 1.4;  
    double f = 8.0;  
    var x = 10;  
    var y = "Hallo";  
}
```

Typinferenz mit var wird seit Java 9 unterstützt

# Deklaration von Variablen

## ■ Weitere Regeln

- Variablen müssen initialisiert werden, bevor ihr Wert gelesen werden darf
- mit **final** gekennzeichnet sind es Konstante
  - müssen genau 1x initialisiert werden
  - dürfen im Nachhinein nicht geändert werden

```
String name;  
System.out.println(name); // Compiler-Fehler  
  
final int anzahl;  
anzahl = 5;  
anzahl++; // Compiler-Fehler
```

# Operatorn

## ■ Arithmetische

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo

## ■ Vergleich

==	gleich
!=	ungleich
<	kleiner
>	größer
<=	kleiner oder gleich
>=	größer oder gleich

## ■ In-, Dekrement

++	Inkrement
--	Dekrement

## ■ Bitweise

&	UND
	inklusives ODER
^	exklusives ODER
~	Komplement

## ■ Logische

&&	logisches UND
	logisches ODER
!	logisches NOT

# Operatoren

## ▪ Shift

<code>&lt;&lt;</code>	nach links
<code>&gt;&gt;</code>	nach rechts (signed)
<code>&gt;&gt;&gt;</code>	nach rechts (unsigned)

## ▪ Diverse

<code>.</code>	Memberzugriff
<code>[]</code>	Indexzugriff
<code>? :</code>	Bedingte Bewertung
<code>(Typ)</code>	type cast

## ▪ Zuweisung

<code>=</code>	Zuweisung
<code>+= -= *= /= %=</code>	Abkürzung für arithmetische Operatoren
<code>&lt;&lt;= &gt;&gt;= &gt;&gt;&gt;=</code>	Abkürzung für Shiftoperatoren
<code>&amp;=  = ^=</code>	Abkürzung für Bitoperatoren

# Logische Operatoren

Operation	Ausdruck a	Ausdruck b	Ergebnis
!a (NOT)	false true		true false
a && b (AND)	false false true true	false true false true	false false false true
a    b (OR)	false false true true	false true false true	false true true true
a ^ b (XOR)	false false true true	false true false true	false true true false

^ ist eigentlich der bitweise XOR Operator. Mit zwei boolean Operanden entspricht es einem logischen XOR.

# Inkrement: Post- und Präfix

```
...  
  
public static void main(String args[])
{
    int a=0, b=1, c=2;  
  
    a = b++;          //jetzt: a == 1, b == 2  
  
    c = ++b;          //jetzt: c == 3, b == 3  
  
    while(c < 10)
        System.out.print(c++);  
  
}
```

3456789 wird  
ausgegeben

# Operator Prioritäten

höchste

niedrigste

Beschreibung	Operator
Access, Parentheses	<code>[] . ()</code>
Postfix	<code>a++ a--</code>
Unary	<code>++a --a +a -a ~ !</code>
Cast, Creation	<code>(type) new</code>
Multiplicative	<code>* / %</code>
Additive	<code>+ -</code>
Shift	<code>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</code>
Relational	<code>&lt; &gt; &lt;= &gt;= instanceof</code>
Equality	<code>== !=</code>
Bitwise AND	<code>&amp;</code>
Bitwise XOR	<code>^</code>
Bitwise OR	<code> </code>
Logical AND	<code>&amp;&amp;</code>
Logical OR	<code>  </code>
Ternary	<code>? :</code>
Assignment	<code>= += -= *= /= %= &amp;= ^=  = &lt;&lt;= &gt;&gt;= &gt;&gt;&gt;=</code>

# Standard IO

## ▪ Konsolenausgabe

- System.out und System.err (beide Typ PrintStream)
  - repräsentieren die Standard-Ausgabe und Standard-Fehlerausgabe
- Ausgabe von primitiven Datentypen und Zeichenfolgen
  - print: einen Wert ausgeben
  - println: einen Wert mit Zeilenumbruch ausgeben

```
System.out.print("Text ");
System.out.println("Text mit Zeilenumbruch");
int zahl = 10;
System.out.print(zahl); // Ganzzahl
System.out.println(zahl); // Ganzzahl mit Zeilenumbruch
// Verkettung von Zeichenfolge und Ganzzahl -> Zeichenfolge
System.out.println("Zahl1: " + zahl);
```

```
Text Text mit Zeilenumbruch
1010
Zahl1: 10
```

# Standard IO

## ▪ Formatierte Ausgabe

- printf: Formatierung mit Formatzeichenfolge, Platzhaltern und Argumenten
- Platzhaltersyntax

`%[index$][flags][width].[precision]conversion`

Argument-Index

Gesamtlänge in Zeichen

Darstellung als Text, Zahl, ...

Spezielle Funktionalität  
(je nach Conversion)

Anzahl der  
Nachkommastellen

- Darstellung von Zahlen erfolgt mit Regionaleinstellungen
- `IllegalFormatConversionException`

- tritt auf wenn Conversion und Argumenttyp nicht zusammenpassen, z.B. `d != java.lang.Double`

Conversion d passt nicht zum Argumenttyp Double

# Standard IO

## ▪ Formatierte Ausgabe

```
String s = "Hey!";
char c = 'a';
int i = 90;
double v = 5.678;
```

conversion	Darstellung als
s	Zeichenfolge
c	Unicode-Zeichen
d	Ganzzahl dezimal
x, X	Ganzzahl hexadezimal
f	Fließkommazahl
b	Boolean (true oder false)
%	Prozentzeichen

```
System.out.printf("[%s]\n", s);
System.out.printf("[%6s]\n", s);
System.out.printf("[% -6s]\n", s);
System.out.printf("%c %c\n", c, i);
System.out.printf("%1$d %1$x\n", i, i);
System.out.printf("%04d\n", i);
System.out.printf("%f\n", v);
System.out.printf("%.2f\n", v);
System.out.printf("%05.2f\n", v);
```



```
[Hey!]
[ Hey! ]
[Hey! ]
a z
90 5A
0090
5,678000
5,68
05,68
```

# Standard IO

## ▪ Konsoleneingabe

- `System.in` (Typ `InputStream`)
  - repräsentiert die Standard-Eingabe
  - Methoden liefern Bytes -> Umwandlung erforderlich
- Klasse `Scanner`
  - liest Strings und primitive Typen aus dem Konsoleninput:
    - Leerzeichen, Tab, Zeilenumbruch sind Trennzeichen
    - `next()`, `nextLine()` für String
    - `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, `nextBoolean()`
    - `next().charAt(0)` oder `nextLine().charAt(0)` für char
  - Prüfmethoden
    - `hasNext()`, `hasNextInt()`, `hasNextDouble()`, ...
  - Zeilenumbruch
    - `nextLine` liest den Zeilenumbruch aus dem Puffer
    - alle anderen `next...` Methoden lassen den Zeilenumbruch im Puffer
      - » Achtung: abwechselndes Verwenden von `nextLine` und `next...` kann zu Problemen führen

```
Scanner input =  
    new Scanner(System.in);
```

# **Bedingte Anweisung – if**

```
if (Boole'scher Ausdruck)
    Anweisung oder Anweisungsblock
```

```
if (Boole'scher Ausdruck)
    Anweisung oder Anweisungsblock
else
    Anweisung oder Anweisungsblock
```

```
int x;
...
if (x > 0)
    System.out.println("Positiv!");
else
{
    System.out.println("Negativ!");
    x = -x;
}
```

# Fallunterscheidung – switch Anweisung

```
switch (Ausdruck) {  
    case Konstante1:  
    case Konstante2:  
        Anweisung(en)  
    case Konstante3:  
        Anweisung(en)  
        break;  
    default:  
        Anweisung(en)  
        break;  
}
```

```
switch (Ausdruck) {  
    case Konstante1,  
        Konstante2 ->  
        Anweisung(en)  
    case Konstante3 ->  
        Anweisung(en)  
    default ->  
        Anweisung(en)  
}
```

Standardsyntax in allen  
Java-Versionen

Neue strengere Syntax  
ab Java 14

- für ganzzahlige Ausdrücke, Enums (seit Java 5) und Strings (seit Java 7)

# Fallunterscheidung – switch Anweisung

```
String strFarbe = ...;  
String strTyp;  
  
switch (strFarbe) {  
    case "Karo":  
    case "Herz":  
        strTyp = "Rot";  
        break;  
    case "Pik":  
    case "Treff":  
        strTyp = "Schwarz";  
        break;  
    default:  
        strTyp = "unbekannt";  
}
```

break ist syntaktisch nicht zwingend,  
ohne break geht die Ausführung  
im switch einfach weiter

```
String strFarbe = ...;  
String strTyp;  
  
switch (strFarbe) {  
    case "Karo", "Herz" ->  
        strTyp = "Rot";  
    case "Pik", "Treff" ->  
        strTyp = "Schwarz";  
    default ->  
        strTyp = "unbekannt";  
}
```

Neue Syntax erfordert kein break:  
nach der jeweiligen Anweisung  
wird das switch automatisch  
verlassen

# Fallunterscheidung – switch Ausdruck

- **Seit Java 14**

- Verwendung von switch als Ausdruck möglich
- Syntax existiert ebenfalls in 2 Varianten

```
String strFarbe = ...;
String strTyp =
    switch (strFarbe) {
        case "Karo", "Herz":
            yield "Rot";
        case "Pik", "Treff":
            yield "Schwarz";
        default:
            yield "unbekannt";
    };
```

```
String strFarbe = ...;
String strTyp =
    switch (strFarbe) {
        case "Karo", "Herz" ->
            "Rot";
        case "Pik", "Treff" ->
            "Schwarz";
        default ->
            "unbekannt";
    };
```

# Iterationen – while

- **while-Schleife**

```
while (Boole'scher Ausdruck)
      Anweisung oder Anweisungsblock
```

```
int i;
...
while (i < 10) {
    .....
}
```

- Anweisung bzw. Block wird 0 bis n Mal ausgeführt

# Iterationen – do ... while

- **do-while-Schleife**

```
do
    Anweisung oder Anweisungsblock
  while (Boole'scher Ausdruck);
```

```
int monat;
...
do {
    ....;
} while ( monat < 1 || monat > 12 );
```

- Anweisung bzw. Block wird 1 bis n Mal ausgeführt

# Iterationen – for

## ■ for-Schleife

```
for (<init>;<bedingung>;<aktualisierung>)  
    Anweisung oder Anweisungsblock
```

```
for (int i = 1; i <= 12; i++) {  
    .....;  
}
```

- Anweisung bzw. Block wird 0 bis n Mal ausgeführt
- die Variable i gilt nur in der for-Schleife

# Iterationen – for each

- **for-each-Schleife**

- heißt auch Enhanced for loop

```
for (<declaration> : <expression>)  
    Anweisung oder Anweisungsblock
```

```
double[] zahlen = {3.14, 2.21, 89.9};  
...  
for (double zahl : zahlen) {  
    .....;  
}
```

- Anweisung bzw. Block wird 0 bis n Mal ausgeführt
  - die Variable s gilt nur im Block der for-each-Schleife

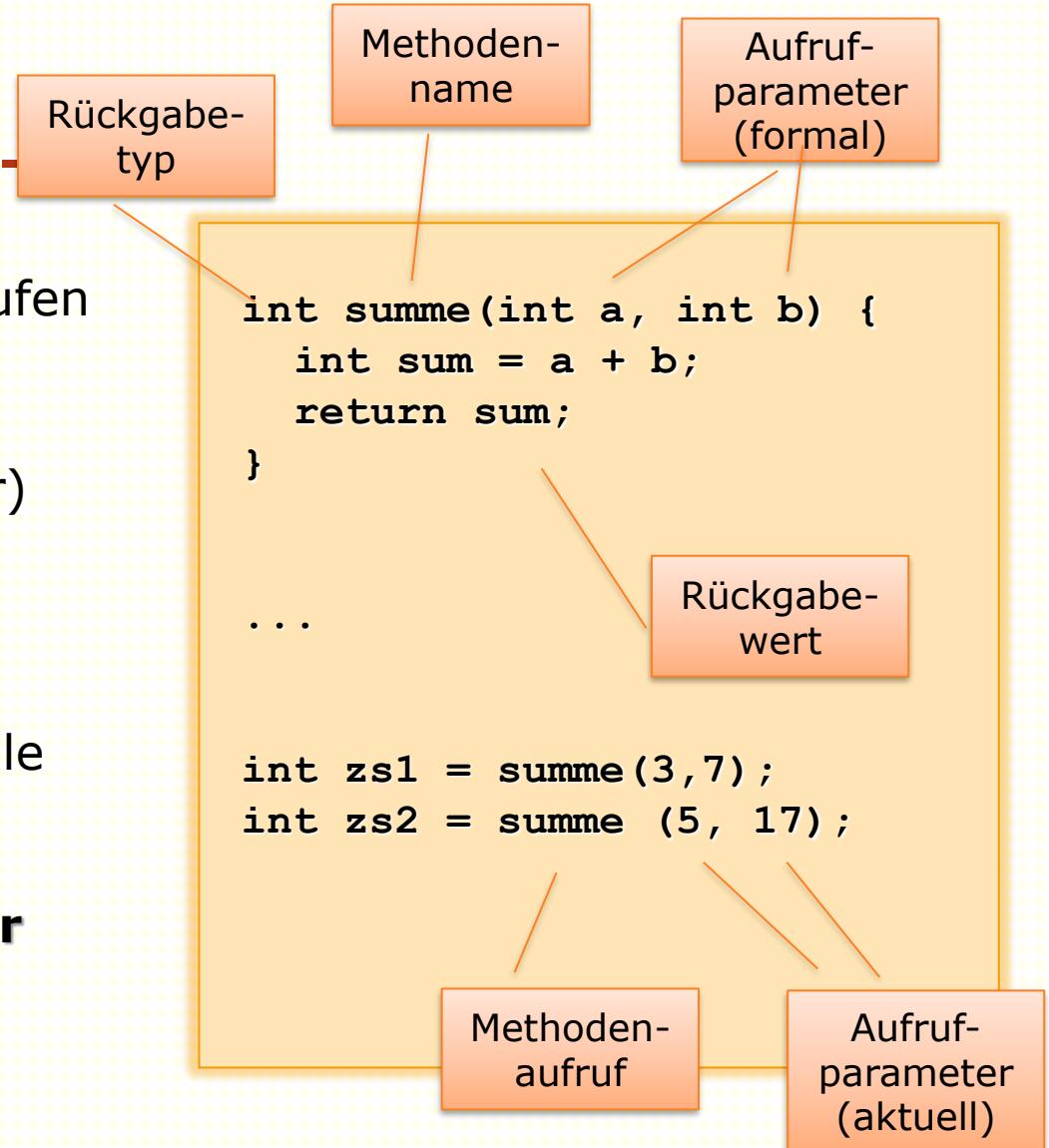
# **Kontrollstrukturen - Sprünge**

- **break**
  - switch-Statement oder Schleife verlassen
- **continue**
  - den nächsten Schleifendurchlauf beginnen
    - while, do... while: Auswertung der Bedingung
    - for, for-each: Reinitialisierung
- **return**
  - die Methode beenden und zum Aufrufer zurück kehren

# Methoden

- **Definiert einen Unter-Algorithmus**

- kann mehrfach aufgerufen werden
- kann beim Aufruf Argumente (Parameter) erhalten
- kann einen Wert zurückliefern
- Die formale Schnittstelle (Rückgabetyp, Name, Parametertypen) heißt **(Methoden-)Signatur**



# Methoden überladen (overload)

- **mehrere Methoden**

- haben denselben Namen
- aber Unterschiede in der Parameterliste:
  - Anzahl der Parameter
  - Typen der Parameter an einer Position

```
int summe(int a, int b) {  
    return a + b;  
}  
int summe(int a, int b, int c) {  
    return a + b + c;  
}  
...  
int s1 = summe(20, 12);  
int s2 = summe(13, 17, 25);
```

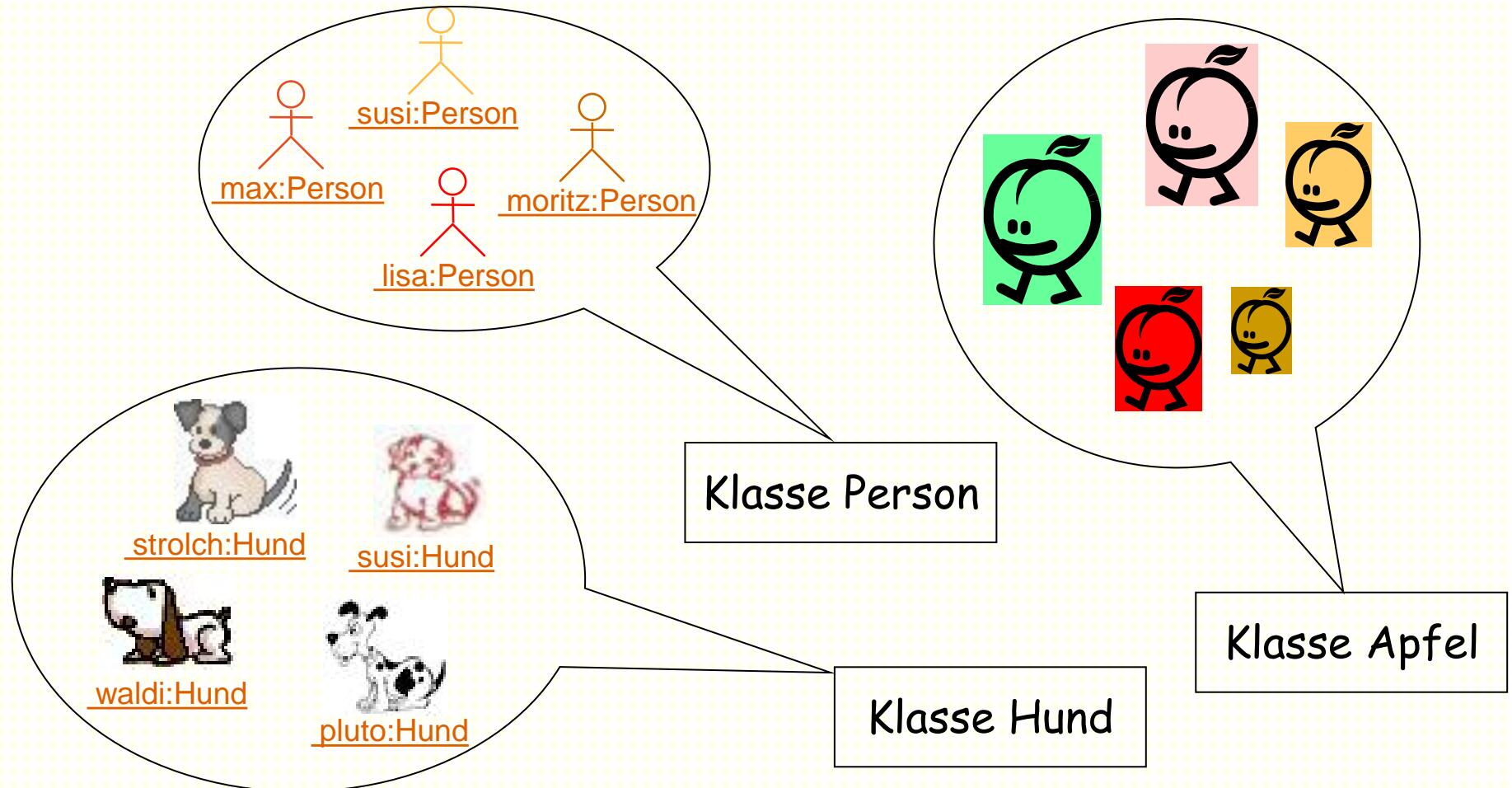
Überladungen der  
Methode "summe"

Compiler unterscheidet je  
nach Anzahl/Typen der  
aktuellen Parameter

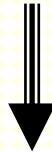
# ***Objektorientierte Konzepte***

Grundlagen

# Klassen und Objekte

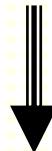


**Klasse  
(Class)**



**Typ**

**Objekt  
(Object)**



**Wert**

- Datentyp
- Enthält Attribute (Daten) und Methoden (Funktionen)
- Schablone für Objekte

- Exemplar ("Instanz") einer Klasse ("Wert")

# *Klassen und Objekte - Klasse*

- **Eine Klasse**
  - beschreibt eine Menge von Objekten
    - mit gleicher Struktur (Attributen) und
    - gleichem Verhalten (Methoden)
- **Jede Klasse**
  - hat einen Namen; dieser ist ein (zusammengesetztes) Hauptwort und beginnt mit großem Anfangsbuchstaben
- **In einer Klasse werden**
  - Attribute (=Daten, Zustand, Status) und
  - Methoden (=Funktionalität, Verhalten) definiert

# Klassendefinition

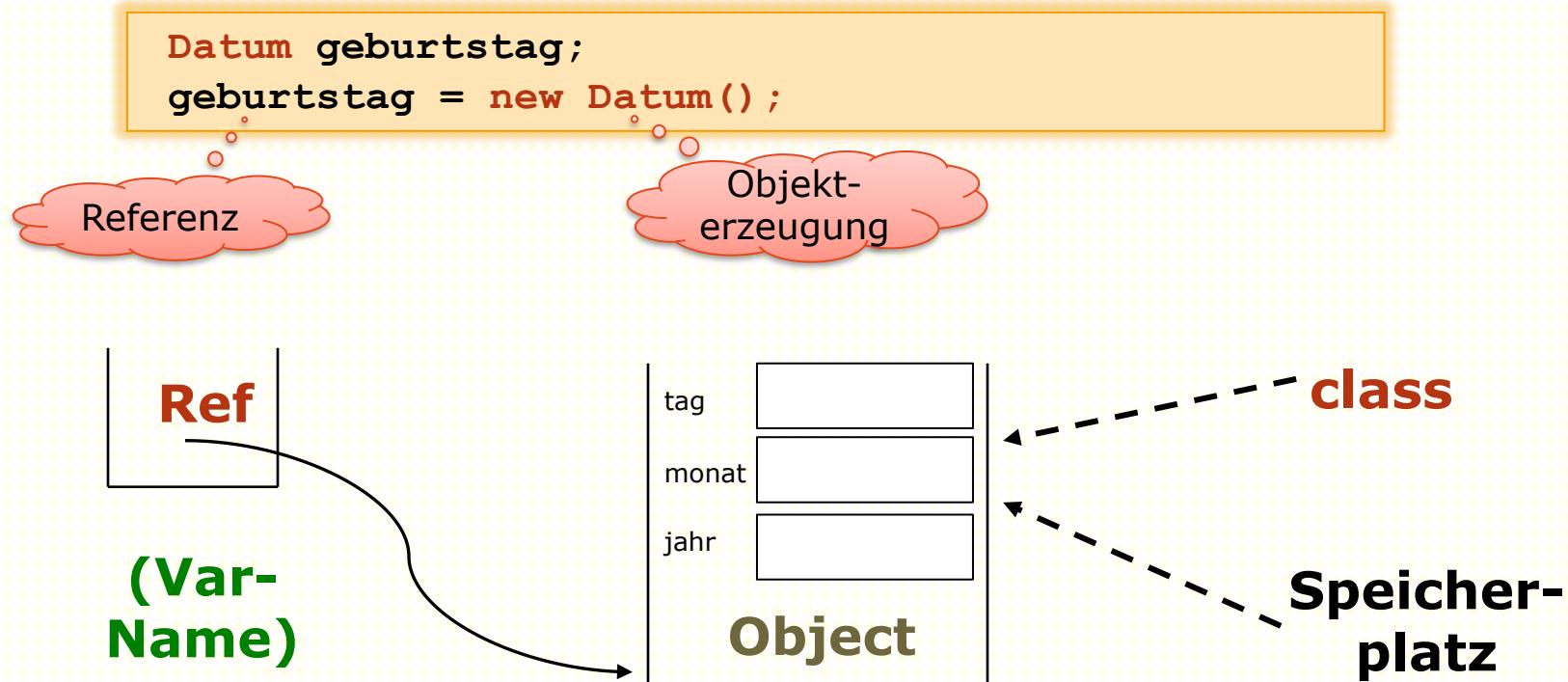
```
public class Datum {  
    // Attribute für Tag, Monat und Jahr  
    private int tag, monat, jahr;  
    // ein Datum setzen  
    public void setzen(int t, int m, int j) {  
        tag = t;  
        monat = m;  
        jahr = j;  
    }  
    // das Datum anzeigen  
    public void ausgeben() {  
        System.out.printf("%02d.%02d.%04d", tag, monat, jahr);  
    }  
    public int calcDiff(Datum start) {  
        .....  
        return ...;  
    }  
}
```

Attribute (oft "private"), legen fest welche Daten die Objekte haben

Methoden (oft "public"), bestimmen welche Aktionen für die Objekte ausgeführt werden können

# Instanziierung von Objekten

- **Erzeugung von Objekten (Instanziierung)**
  - erfolgt in Java ausschließlich dynamisch
  - weitere Verwendung immer über Referenz

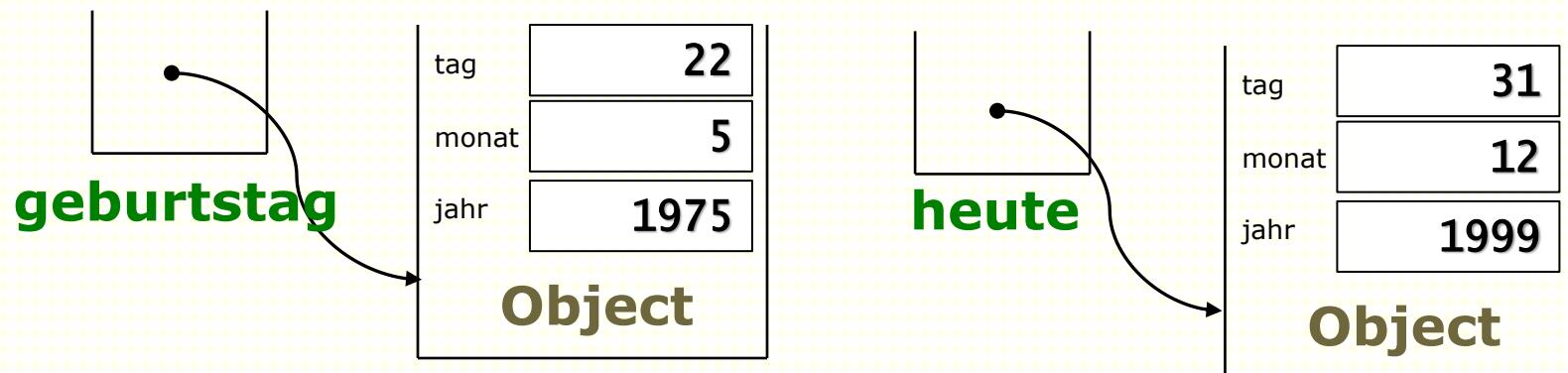


# Verwendung von Objekten

- **Zugriff auf ein Objekt**
  - erfolgt über eine Referenz
  - mit dem Operator für den Memberzugriff `.`

```
Datum geburtstag = new Datum(), heute = new Datum();
geburtstag.setzen(22,5,1975); // Datum setzen
geburtstag.ausgeben(); // Datum anzeigen
heute.setzen(31,12,1999); // Datum setzen
heute.ausgeben(); // Datum anzeigen
int diffTage = heute.calcDiff(geburtstag);
System.out.println(diffTage + " Tage");
```

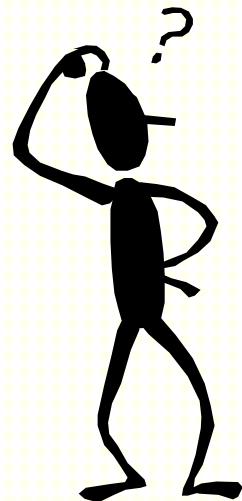
22.05.1975  
31.12.2000  
8989 Tage



# Zugriffsattribute

<b>Memberzugriff</b>	
<b>private</b>	nur die Klasse selbst
<b>package</b>	alle im selben Package; <b>wird nicht angegeben</b>
<b>protected</b>	die Klasse, abgeleitete Klassen und alle im selben Package
<b>public</b>	jeder
<b>Zugriff für Top-Level-Typen (stärker als Memberzugriff)</b>	
<b>package</b>	nur im selben Package verwendbar; <b>wird nicht angegeben</b>
<b>public</b>	überall verwendbar

# Konstruktor



- **In Java muss jedes neue Objekt initialisiert werden**
  - dazu dient ein Konstruktor
- **Ein Konstruktor**
  - ist eine Methode, die beim Erzeugen des Objekts **automatisch aufgerufen** wird
  - heißt so wie die Klasse
  - hat **keinen Rückgabetyp** (auch nicht void)
  - kann **überladen** werden
  - kann nicht explizit aufgerufen werden
    - Ausnahme: aus anderem Konstruktor

# Konstruktor

```
public class Datum {  
    private int tag, monat, jahr;  
    public void setzen(int t, int m, int j) {.....}  
    public void ausgeben() {.....}  
    public int calcDiff(Datum start) {....}  
  
    public Datum(int t, int m, int j) {  
        tag = t;  
        monat = m;  
        jahr = j;  
    }  
    public Datum() {  
        this(1, 1, 2000);  
    }  
}  
...  
Datum d1 = new Datum();  
Datum d2 = new Datum(31, 12, 1999);
```

Ein Konstruktor hat  
**keinen** Rückgabetyp

Ein Konstruktor heißt so wie  
seine Klasse: **Datum**

Ein Konstruktor kann  
**überladen** werden

Als 1. Anweisung kann ein **anderer**  
Konstruktor aufgerufen werden

Objekterzeugung ist mit allen  
definierten Konstruktoren möglich

# Konstruktor

## ▪ Automatische Initialisierung

- Bevor der Konstruktor läuft, werden alle Attribute initialisiert
  - mit 0, null oder false
  - bzw. mit den angegebenen Initialwerten

```
public class Datum {  
    private int tag = 1,  
    monat = 1,  
    jahr = 2000;  
    ...  
}
```

## ▪ Defaultkonstruktor

- wird automatisch erzeugt, wenn die Klasse keinen Konstruktor enthält
- hat keine Parameter
- enthält keine Anweisungen

## ▪ Defaultkonstruktor wird nicht erzeugt

- wenn die Klasse irgend einen Konstruktor enthält

# ***Werttypen und Referenztypen***

	<b>Werttypen</b>	<b>Referenztypen</b>
Allokation	am Stack	am Heap
Instanzierung	durch Deklaration	mit new
Zerstörung	am Ende des Blocks	Garbage Collector
Zuweisung	Kopie des Wertes	Kopie der Referenz
Parameter-Übergabe	Kopie des Wertes	Kopie der Referenz
Return-Wert	Kopie des Wertes	Kopie der Referenz
Betrifft	primitive Typen	Klassen, Enums, Interfaces

# Werttypen und Referenztypen

- **Kopieren einer primitiven Variable**
  - Kopiert den Wert
- **Kopieren einer Variable einer Klasse**
  - Kopiert die Referenz, nicht das Objekt

```
int zahl1, tmpZahl;  
zahl1 = 10;  
tmpZahl = zahl1;
```

**zahl1**

10

**tmpZahl**

10

```
Datum geburtstag, tempDat;  
geburtstag = new Datum(22, 5, 1975);  
tmpDat = geburtstag;
```

**geburtstag**

tag	22
monat	5
jahr	1975

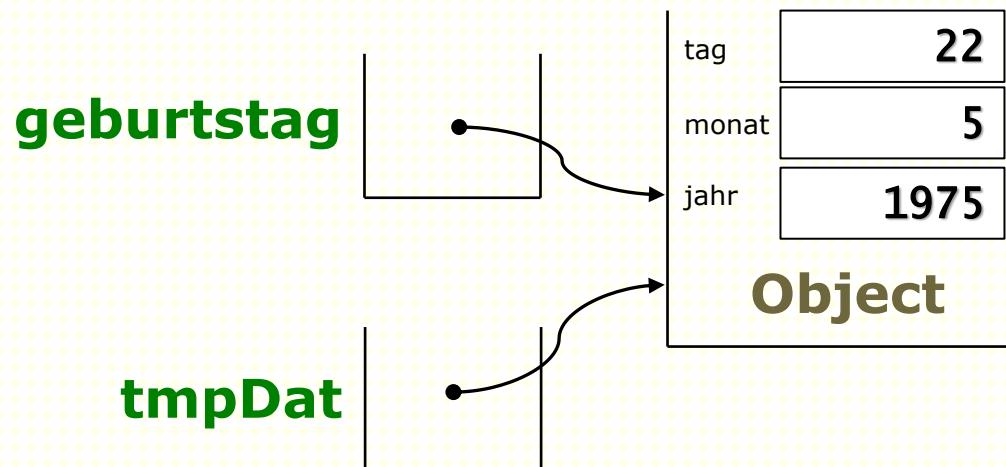
**tmpDat**

**Object**

# Werttypen und Referenztypen

- **Kopieren einer primitiven Variable**
  - Kopiert den Wert
- **Kopieren einer Variable einer Klasse**
  - Kopiert die Referenz, nicht das Objekt

```
Datum geburtstag, tempDat;  
geburtstag = new Datum(22, 5, 1975);  
tmpDat = geburtstag;
```



# Statische Felder und Methoden

```
class CountClass {  
    private static int objectCount;  
  
    private int id;  
  
    public CountClass(int id) {  
        this.id = id;  
        objectCount++;  
    }  
  
    public int GetId() {  
        return /*this.*/id;  
    }  
  
    public static int GetObjectCount() {  
        return /*CountClass.*/objectCount;  
    }  
}
```

Das static Feld objectCount gibt es nur einmal für die ganze Klasse!

Jedes Objekt hat Speicherplatz für den Wert des Instanzfeldes id

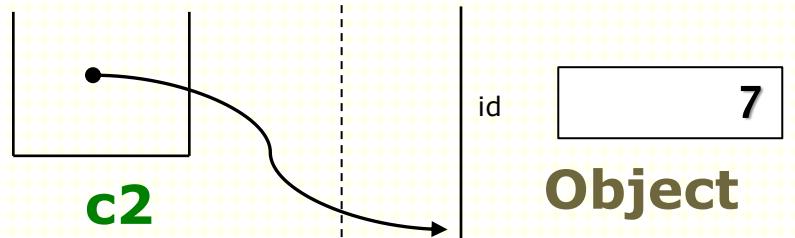
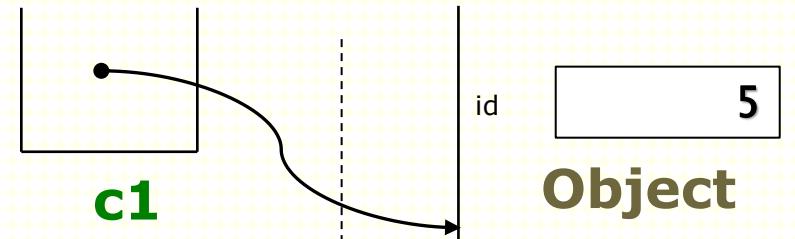
In Instanzmethoden gibt es eine implizite Referenz auf das aktuelle Objekt: this

In static Methoden kann nur auf static Elemente zugegriffen werden

# Instanz- vs. statische Member

```
public void test(){  
    CountClass c1 = new CountClass(5);  
    CountClass c2 = new CountClass(7);  
    int id1 = c1.GetId(), id2 = c2.GetId();  
    int count = CountClass.GetObjectCount();  
}
```

Der Zugriff auf Instanzmember erfolgt über eine Referenz



Stapelspeicher  
(Stack)

dynamischer  
Speicher (Heap)

Der Zugriff auf statische Member erfolgt über den Klassennamen

0 1 2

**CountClass.  
objectCount**

statischer Speicher

# *Instanz- vs. Statische Member*

	<b>Instanzmember</b>	<b>Statische Member</b>
Zugehörigkeit	sind an ein Objekt gebunden	gehören direkt zur Klasse
Verwendung	es muss zuvor ein Objekt erzeugt worden sein	es ist kein Objekt erforderlich
Zugriff	nur über eine Referenz	ohne Referenz, über den Klassennamen
Kennzeichnung	keine (ist der Normalfall)	mit Keyword <b>static</b>
Einsatz	<ul style="list-style-type: none"><li>○ mehrere Exemplare der Klasse mit jeweils eigenen Werten;</li><li>○ Nutzen fortgeschritten OOP-Techniken</li></ul>	<ul style="list-style-type: none"><li>○ Utility-Methoden, die ohne vorherige Instanziierung verwendbar sein sollen;</li><li>○ "Globale" Methoden</li></ul>

# **Initialisierung von static Feldern**

- **Static Initializer**

- für die Initialisierung von static Feldern
- automatischer Aufruf vor der Instanziierung des 1. Objekts bzw. vor dem 1. Zugriff auf static Members

```
public class AutoId {  
    private static int nextId;  
    static {  
        nextId = 4711;  
    }  
    private int id;  
    public AutoId() {  
        id = nextId++;  
    }  
    ...  
}
```

# **final für Attribute**

- **Unveränderliche Felder**
  - werden als **final** gekennzeichnet
  - müssen genau 1x initialisiert werden:
    - Instanzfelder
      - mit Feldinitialisierung oder
      - im Konstruktor
    - Statische Felder
      - mit Feldinitialisierung oder
      - im Static Initializer
  - Statische final Felder werden als globale Konstante verwendet

```
public class Datum {  
    public final static int MIN_JAHR = 1602;  
    ...  
}
```

# Aufzähltyp – enum

- **Spezielle Art von Klasse**
  - es gibt nur die im Enum definierten Instanzen
  - nützliche Methoden
    - `toString`: liefert den Namen der Instanz
    - `valueOf`: liefert die Instanz zum Namen
    - `ordinal`: liefert den Ordinalwert der Instanz
  - kann im switch verwendet werden

```
public enum Wochentage{  
    // die definierten Instanzen  
    MONTAG, DIENSTAG, MITTWOCH,  
    ...;  
}  
...  
Wochentage tag1 =  
    Wochentage.MONTAG;  
...
```

```
Wochentage wTag =  
    Wochentage.valueOf(...);  
switch(wTag) {  
    case MONTAG:  
    case DIENSTAG:  
        System.out.print("Juhu");  
        break;  
}
```

- **Java-Klasse, die sich an bestimmte Kodierungs-Richtlinien hält**
  - enthält öffentlichen Defaultkonstruktor
  - Properties
    - Eigenschaften, über die jedes Objekt der Klasse verfügt
    - werden mit get/is- und set-Zugriffsmethoden implementiert
    - können unabhängig von dahinterliegenden Attributen implementiert werden
  - andere Methoden
    - definieren beliebige weitere Funktionalität

# Java Bean - Property

```
public class Bankkonto {  
    private String strInhaber;  
    public String getInhaber () {  
        return strInhaber;  
    }  
    public void setInhaber (String inhaber) {  
        this.strInhaber = inhaber;  
    }  
  
    private int knr;  
    public int getKontoNummer () {  
        return knr;  
    }  
}
```



Property  
"inhaber"  
(Typ String)



Readonly Property  
"kontoNummer"  
(Typ int)

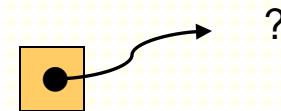
# Array – Vektor

- **Zusammenhängender Block von Werten desselben Typs**
  - Referenztyp
    - Instanziierung mit `new` Operator
  - Anzahl der Elemente: `length` (final Feld)
  - Zugriff auf Elemente: per `Index` (beginnt bei 0)
  - Iteration mit `for-each` wird unterstützt
- **Arrays Klasse**
  - unterstützende Funktionalität für (eindimensionale) Arrays
    - `sort`, `binarySearch`, `toString`, ...

# Array - eindimensional

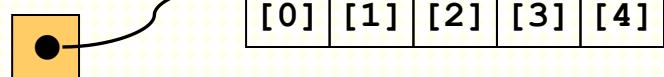
```
int[] zahlen;
```

Deklaration



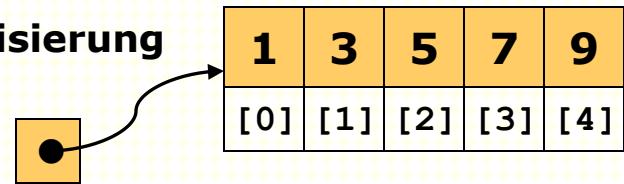
```
zahlen = new int[5];
```

Erzeugung



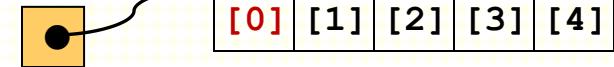
```
zahlen = new int[] { 1, 3, 5, 7, 9 };  
int[] zahlen = { 1, 3, 5, 7, 9 };
```

Initialisierung



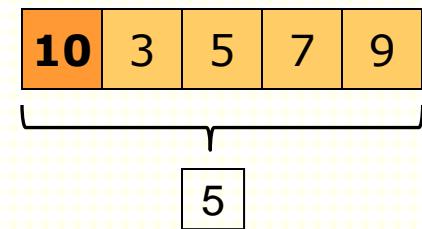
```
zahlen[0] = 10;
```

Zugriff per Index



```
int len = zahlen.length;
```

Anzahl der Elemente



# Array – zweidimensional

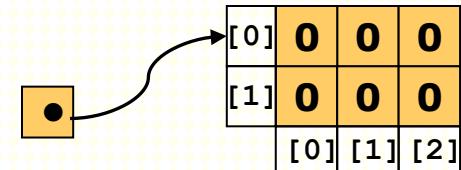
```
int[][] zahlen;
```

Deklaration



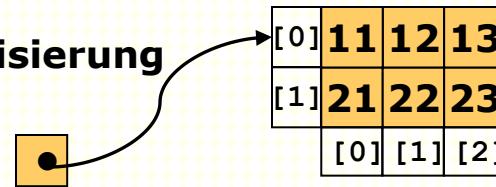
```
int[][] zahlen = new int[2][3];
```

Erzeugung



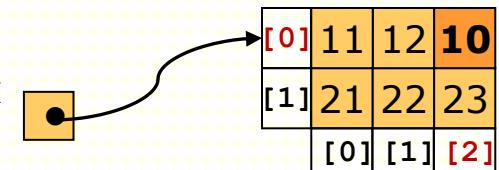
```
int[][] zahlen =  
{ { 11, 12, 13 },  
  { 21, 22, 23 } };
```

Initialisierung



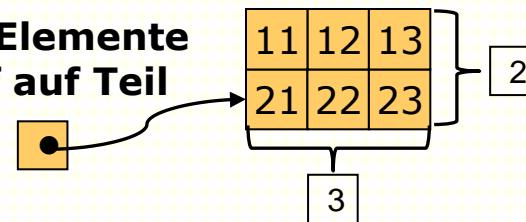
```
zahlen[0][2] = 10;
```

Zugriff per Index



```
int len = zahlen.length; // 2  
int [] row = zahlen[1];
```

Anzahl Elemente  
Zugriff auf Teil



# Wrapperklassen

- **Hilfsklasse pro primitivem Typ**
  - Byte, Short, Integer, Long, Float, Double, Character, Boolean
  - Viele nützliche Methoden für den jeweiligen primitiven Typ, meist static
  - Gemeinsame Funktionalität
    - valueof: liefert ein Wrapper-Objekt zu einem primitivem Wert oder aus einem String
    - toString: Zeichenfolgendarstellung für primitiven Wert
  - Funktionalität für Zahlentypen
    - Konstante für min./max. Wert
    - toHexString, toBinaryString (nur für ganzzahlige): weitere Umwandlungsmethoden nach String
    - parseXXX-Methode je nach Typ (parseInt, parseDouble ...): primitiven Wert aus String ermitteln

# Wrapperklassen

## ▪ Hilfsklassen

- Double, Float
  - Konstante für bestimmte double Werte: NaN, POSITIVE\_INFINITY
  - isNaN, isInfinite, isFinite (Prüfmethoden)
- Character
  - isLetter, isDigit, isSpaceChar, isUpperCase, isLowerCase, ... (Prüfmethoden)
  - toUpperCase, toLowerCase: Umwandlung in Groß- bzw. Kleinbuchstaben
- Boolean
  - parseBoolean: einen boolean aus einem String ermitteln

# Zeichenfolgen

- **Klasse String**
  - kapselt eine unveränderliche Unicode-Zeichenfolge

Methode	Zweck
length	Ermitteln der Länge
charAt	Zeichen (char) an Indexposition ermitteln
indexOf, lastIndexOf	die Indexposition eines Zeichens ermitteln (-1 wenn das Zeichen nicht vorkommt)
equals, equalsIgnoreCase	die Instanz mit einer anderen Zeichenfolge vergleichen
contains, startsWith, endsWith,	prüfen ob die Zeichenfolge eine andere enthält, mit ihr startet oder endet
isEmpty, isBlank	prüfen ob die Zeichenfolge leer ist oder nur Whitespace Zeichen-enthält

Achtung: == und != vergleichen die Referenzen

# Zeichenfolgen

- **Klasse String**
  - Methoden, die eine neue Zeichenfolge erzeugen

Methode	Zweck
toLowerCase, toUpperCase	eine Zeichenfolge in Klein- bzw. Großbuchstaben umwandeln
replace	ein Zeichen durch ein anderes ersetzen (oder eine Zeichenfolge durch eine andere)
trim, strip	Whitespace-Zeichen vorne und hinten abschneiden
stripLeading, stripTrailing	Whitespace-Zeichen vorne / hinten abschneiden (seit Java 11)
substring	Teilzeichenfolge ermitteln
concat	verkettet zwei Zeichenfolgen
+, +=	Operator für Zeichenfolgenverkettung

# Zeichenfolgen

## ▪ Formatierung

- static format: Zeichenfolge aus Format-String, Platzhaltern und Argumenten erzeugen
- formatted: Zeichenfolge aus Format-String-Instanz, Platzhaltern und Argumenten erzeugen
- Platzhalter und Argumente analog zu printf

```
int tag = 1, monat = 12, jahr = 1999;  
String strDat1 = String.format("%02d.%02d.%04d",  
    tag, monat, jahr);  
// oder  
String formatString = "%02d.%02d.%04d";  
String strDat2 = formatString.formatted(tag, monat, jahr);
```



01.12.1999

# Zeichenfolgen

- **Umwandlung String – primitive Typen**
  - `String.valueOf`: Zeichenfolge für primitiven Wert ermitteln
  - `Integer.parseInt`, `Double.parseDouble`, ...: primitiven Wert aus einer Zeichenfolge ermitteln
  - Zahlenformate entsprechen der Java-Norm
    - dh Kommazeichen ist immer `.`

```
double v1 = 234.78;  
String s1 = String.valueOf(v1); // => "234.78"  
  
String s2 = "234.78";  
double v2 = Double.parseDouble(s2); // => 234.78d
```

## ▪ Umwandlung String - Zahlentypen

### – Klasse NumberFormat

- unterstützt verschiedene Regionaleinstellungen
- statische Methoden liefern vordefinierte Formatinstanzen
  - getNumberInstance: allgemeines Zahlenformat
  - getCurrencyInstance: Währungsformat
  - getPercentInstance: Prozentdarstellung (1.0 entspricht 100%)
- format: Zeichenfolge für eine Zahl ermitteln
- parse: Zahl aus einer Zeichenfolge ermitteln

```
NumberFormat numFmt = NumberFormat.getNumberInstance();  
double v1 = 234.78;  
String s1 = numFmt.format(v1); // => "234,78"  
String s2 = "234,78";  
try {  
    double v2 = numFmt.parse(s2).doubleValue(); // => 234.78d  
} catch (ParseException e) { ... }
```

# Zeichenfolgen

- **Klasse StringBuilder**

- kapselt eine veränderliche Unicode-Zeichenfolge
- der Puffer wird bei Bedarf neu allokiert

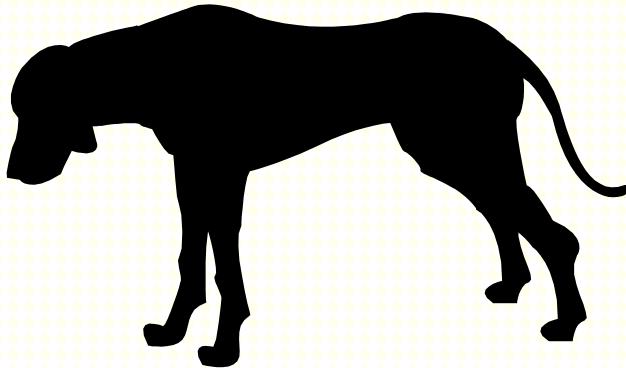
Methode	Zweck
length, charAt, indexOf, lastIndexOf	analog String
append, insert	Zeichenfolge oder primitiven Wert am Ende oder ab Index einfügen
delete, deleteCharAt	Zeichen von-bis bzw. am Index löschen
setLength	neue Länge setzen
replace	Zeichen von-bis durch andere Zeichenfolge ersetzen
reverse	die Zeichenfolge umdrehen
substring	neue Teilzeichenfolge ermitteln

# ***Vererbung***

Spezialisierung von Klassen

# **Vererbung – Ableitung**

- **Die abgeleitete Klasse ist eine Spezialisierung einer (Basis-)Klasse**



**Hund**

- Basisklasse
  - Grundattribute und -methoden von Hunden

Dackel ist  
abgeleitet von  
Hund



**Dackel**

- Wie Hund, aber:
  - einige Dinge anders
  - zusätzliche Funktionalität

# Vererbung

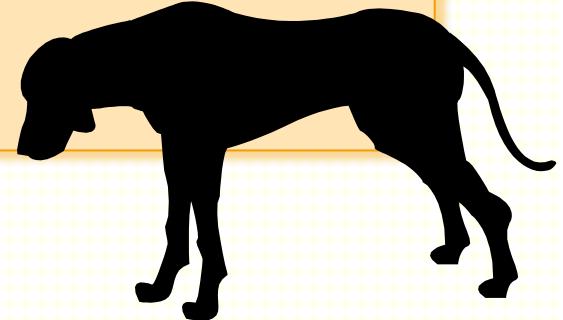
class Hund

Dackel wird von  
Hund abgeleitet

class Dackel extends Hund

Java erlaubt zwischen Klassen  
nur Einfachvererbung

```
public class Hund {  
    private int gewicht;  
    public void setGewicht(int g)  
    {...}  
    ...  
}
```



```
public class Dackel  
    extends Hund {  
    ...  
}  
...  
Dackel waldi = new Dackel();  
waldi.setGewicht(14);
```



# Vererbung – Konstruktor Reihenfolge

- **Beim Erzeugen eines Objekts einer abgeleiteten Klasse wird immer**
  - zuerst der Konstruktor der Basisklasse aufgerufen,
  - dann der Konstruktor der abgeleiteten Klasse

```
public class Hund {  
    ...  
}
```

```
public class Dackel  
    extends Hund {  
    ...  
}
```

```
Dackel waldi = new Dackel();
```



1. Konstruktor von Hund
2. Konstruktor von Dackel

# Vererbung – Konstruktor Reihenfolge

- Der Compiler fügt dafür einen impliziten super-Aufruf im Konstruktor ein

```
public class Hund {  
    ...  
}
```

```
public class Dackel  
    extends Hund {  
  
    public Dackel(){  
        super();  
    }  
    ...  
}
```

```
Dackel waldi = new Dackel();
```



1. Konstruktor von Hund
2. Konstruktor von Dackel

# Vererbung – expliziter super-Aufruf

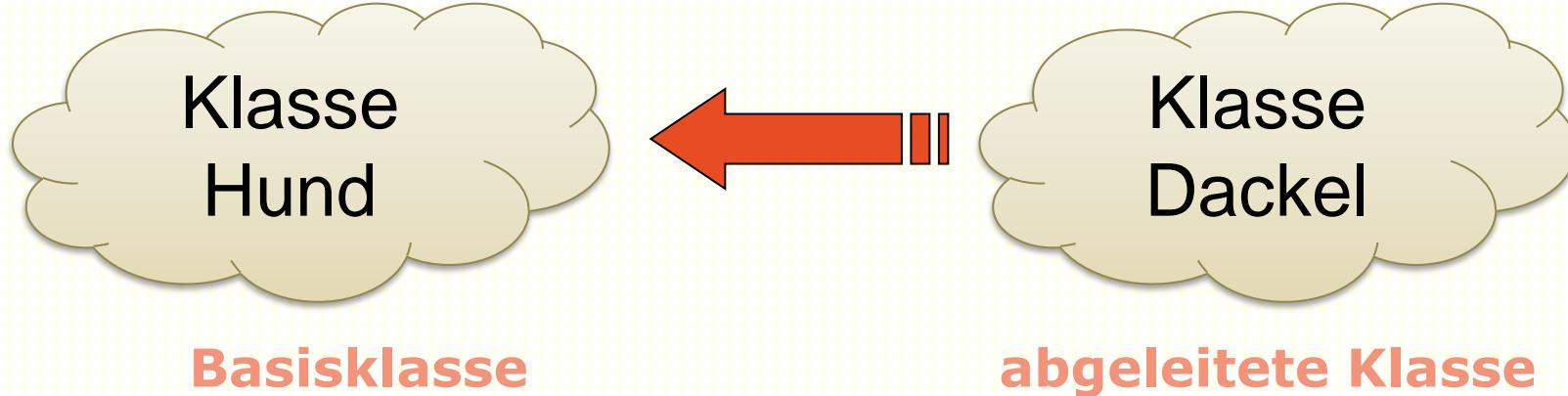
- **Falls der Basisklassen-Konstruktor Parameter hat**
  - werden die Parameter über einen **expliziten super-Aufruf** übergeben
  - Der super-Aufruf muss als **1. Anweisung** in einem Konstruktor stehen

```
public class Hund {  
    public Hund(int gewicht) { ..... }  
}
```

```
public class Dackel extends Hund {  
    public Dackel(int gewicht) {  
        super(gewicht);  
        ...  
    }  
}
```

```
Dackel waldi = new Dackel(14);
```

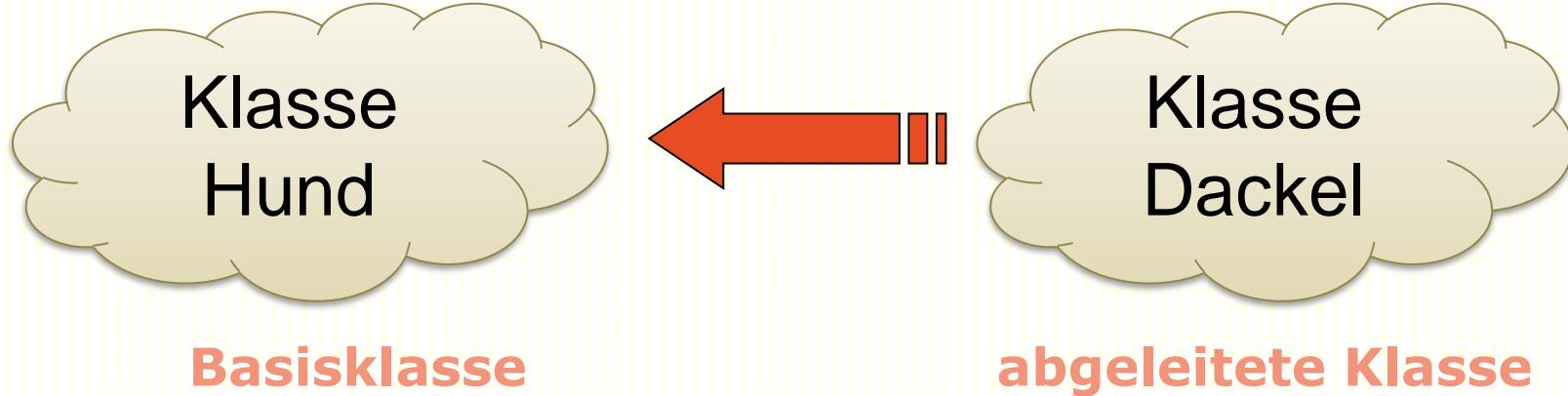
# Vererbung – Typkompatibilität



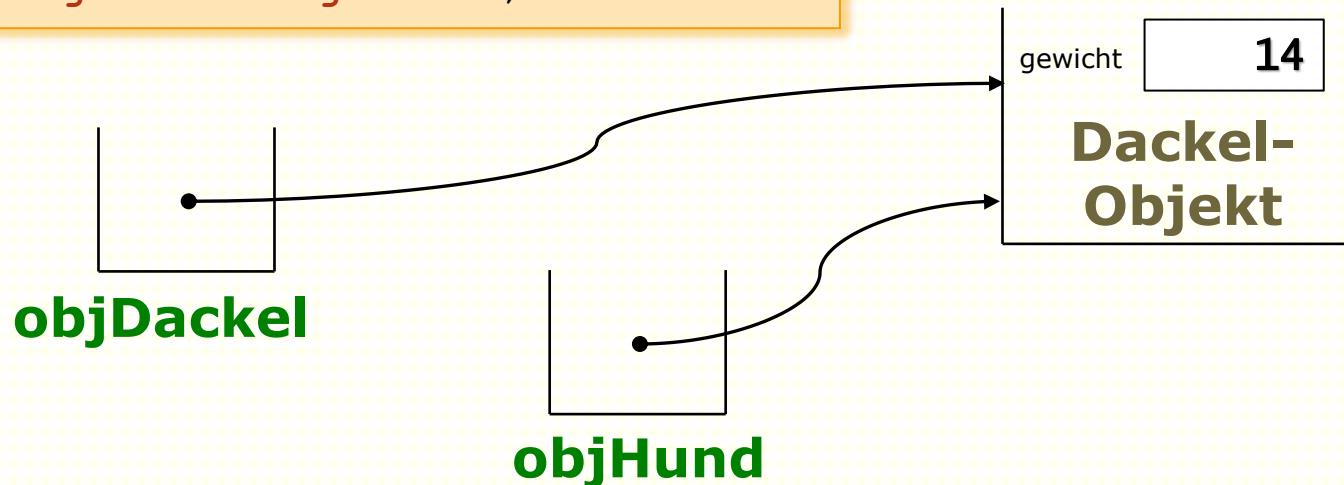
```
Dackel objDackel = new Dackel(14);  
Hund objHund = objDackel;
```

- **Referenz einer abgeleiteten Klasse**
  - kann einer Referenz der Basisklasse zugewiesen werden
- **umgekehrte Zuweisung**
  - ist implizit nicht möglich

# Vererbung – Typkompatibilität

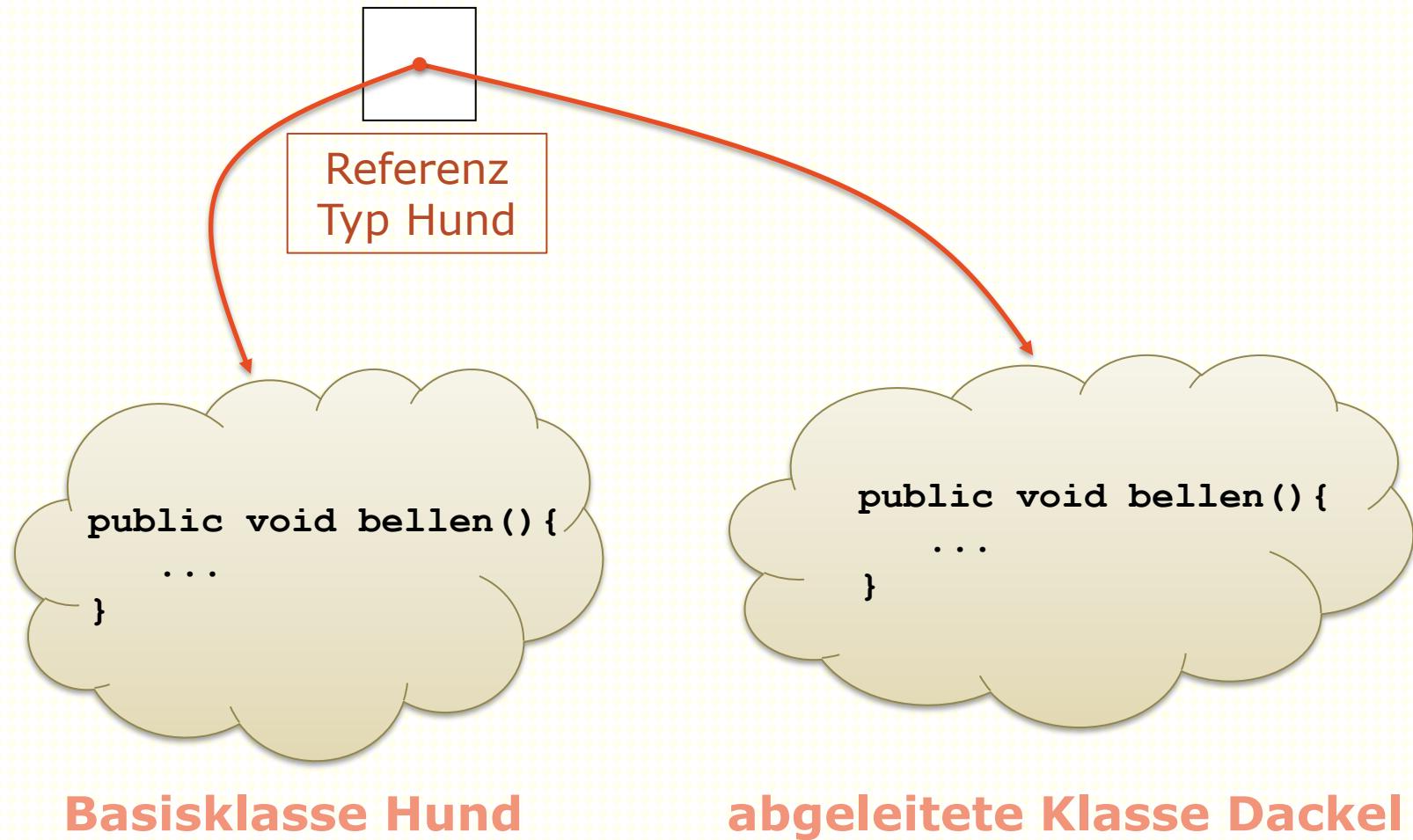


```
Dackel objDackel = new Dackel(14);  
Hund objHund = objDackel;
```



# Polymorphismus

- Welche Funktion wird aufgerufen?



# **Polymorphismus**

## ▪ **Polymorphe Methode**

- Methode der abgeleiteten Klasse **überschreibt** eine Basisklassen-Methode mit der gleichen Signatur
- Zur Laufzeit wird die **zum aktuellen Objekt** gehörende Methode aufgerufen.
  - Dynamisches Binden ("**late binding**")
- Das Objekt kann in **mehreren Gestalten** auftreten (**polymorph**)
- Konstruktoren sind nie polymorph und sollten keine polymorphen Methoden aufrufen!

# Polymorphismus

- **in Java sind Instanzmethoden automatisch polymorph**
  - abgeleitete Klassen überschreiben die Methode, indem sie eine Methode mit derselben Signatur definieren
  - Implementierung der Basisklasse
    - kann über `super` aufgerufen werden
  - Die Annotation `@Override`
    - kennzeichnet die Methode als Überschreibung
    - schützt vor Fehlern durch nicht übereinstimmende Signaturen

```
public class Hund {  
    public void bellen() {  
        ....  
    }  
}
```

```
public class Dackel extends Hund {  
    @Override public void bellen() {  
        super.bellen();  
        ....  
    }  
}
```

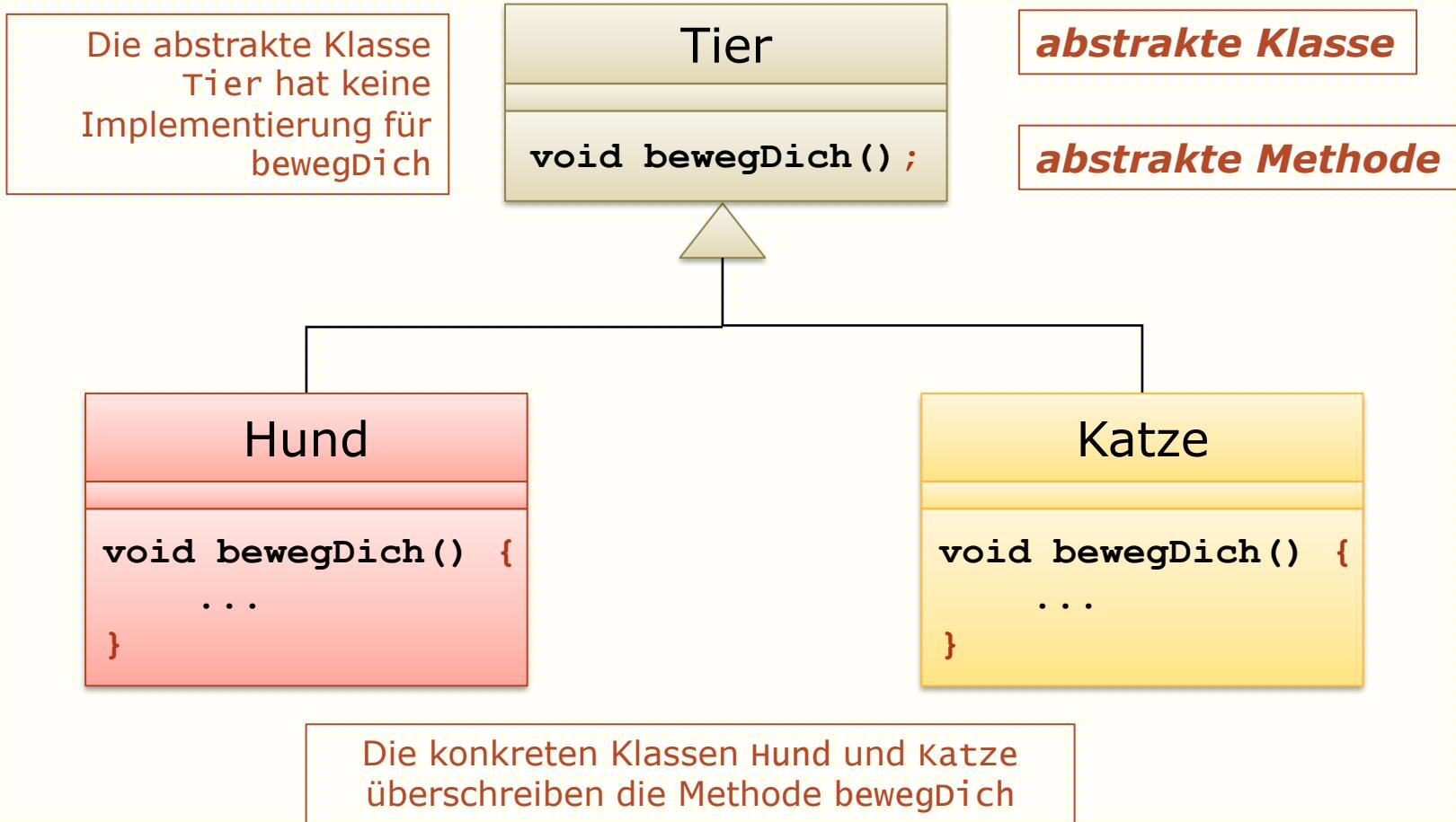
# Polymorphismus

- **Nicht polymorph sind**
  - static Methoden
  - private Methoden (Instanz oder static)
  - mit final gekennzeichnete Instanzmethoden
    - Definieren einer Methode mit derselben Signatur führt zu Compiler Fehler

```
class Hund {  
    protected final void finalMethod() {  
        .....  
    }  
}  
  
class Dackel extends Hund {  
    protected void finalMethod() // Compiler Fehler  
    {  
        .....  
    }  
}
```

# Polymorphismus

## ▪ Abstrakte Klassen und Methoden



# Polymorphismus

## ▪ Abstrakte Methoden

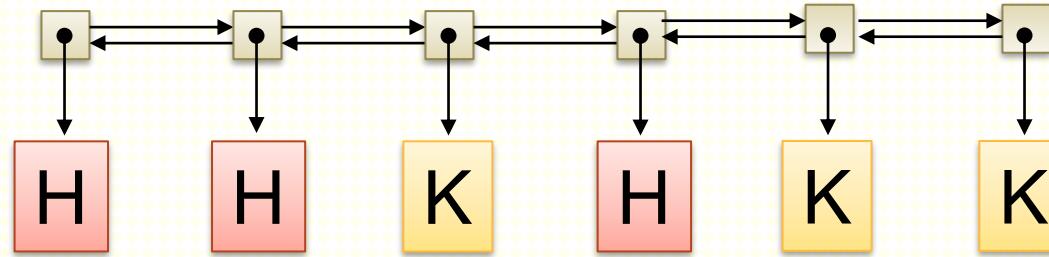
- haben **keine** Implementierung
- werden mit dem Schlüsselwort **abstract** gekennzeichnet
- dürfen nur **in abstrakten** Klassen stehen
- müssen in abgeleiteten Klassen überschrieben werden
- Implementierung der Basisklasse steht **nicht** zur Verfügung

```
abstract class Tier {  
    public abstract void bewegDich();  
}
```

```
class Katze extends Tier {  
    @Override public void bewegDich() {  
        System.out.println("Katzenbuckel");  
    }  
}
```

# Polymorphismus

- **Abstrakte Klasse**
  - kann nicht instanziert werden
  - kann beliebige nicht-abstrakte Member haben
  - definiert **gemeinsame Schnittstelle** für verwandte Klassen
  - vereinfacht die Verwaltung von "verwandten" Objekten
- **Beispiel: Liste von Tier-Objekten**



- Jedes Element "versteht" die Methode `bewegDich`
  - Zur Laufzeit wird dynamisch das richtige **bewegDich** ausgeführt

# Interface

- **definiert eine Schnittstelle**

- enthält
  - abstrakte Methoden
  - Konstanten (final static)
- Methoden sind automatisch
  - **public** und **abstract**
- Implementierung
  - durch Klasse
  - eine Klasse kann mehrere Interfaces implementieren
- Mehrfachvererbung
  - zwischen Interfaces möglich

```
public /*abstract*/ interface Media {  
    /*public abstract*/ void play();  
    /*public abstract*/ String getFilename();  
}
```

# Interface

- **Implementierung durch Klasse**
  - Angabe der Schnittstelle (**implements**)
  - automatisch durch Definition aller Methoden

```
public class Video implements Media {  
    String name;  
    @Override public void play() {  
        ...  
    }  
    @Override public String getFilename() {  
        ...  
    }  
}
```

```
Video meinFilm = new Video();  
Media meinMM = meinFilm;  
meinMM.play();
```

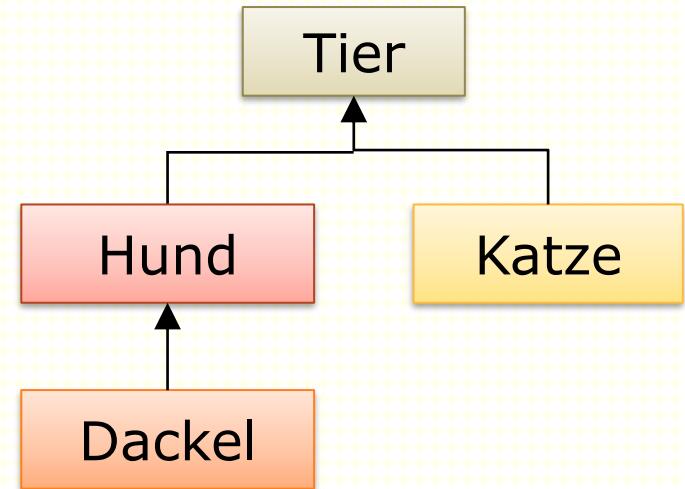
# Interface

- **Grundsatz**
  - "Ein Interface enthält keinen Code" (=keine Implementierung)
- **gilt seit Java 8 nicht mehr**
  - Interfaces dürfen Methoden-Implementierungen enthalten
    - **static** Methode: um eine Hilfsmethode bereitzustellen
    - **default** Methode: um eine Interface-Methode mit Default-Implementierung bereitzustellen
    - dürfen seit Java 9 auch private sein
  - Wird in der neuen Stream API oft angewendet

# Typinformation

## ▪ instanceof

- prüft ob ein Objekt (direkt oder über Vererbung) vom angegebenen Typ ist
  - für Klassen und Interfaces

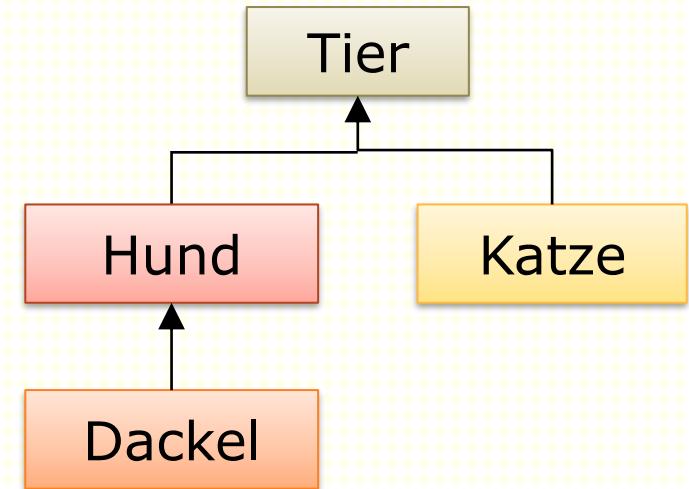


```
void tuWas(Tier t) {  
    // wenn t vom Typ Hund ist, ist  
    // explizite Typumwandlung möglich  
    if (t instanceof Hund) {  
        Hund h = (Hund)t;  
        h.belle();  
    }  
}
```

```
...  
tuWas(new Hund());  
tuWas(new Katze());  
tuWas(new Dackel());  
...
```

# Typinformation

- **Typ-Objekt (Class)**
  - class
    - liefert das Typ-Objekt zu einer Klasse
  - getClass()
    - liefert das Typ-Objekt zu einem Objekt



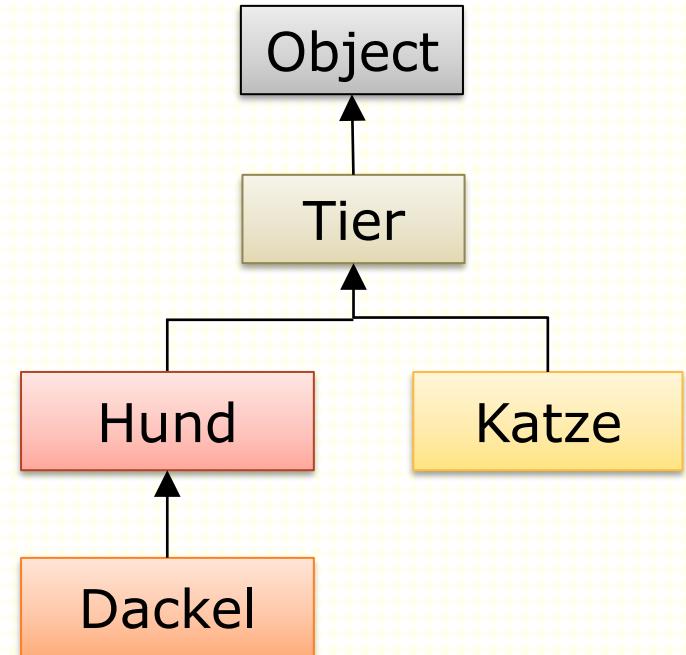
```
void tuWas(Tier t) {  
    System.out.println(t.getClass().getName());  
    // wenn es exakt der Typ Hund ist  
    if (t.getClass() == Hund.class) {  
        System.out.println("Exakt Typ Hund");  
    }  
}
```

```
...  
tuWas(new Hund());  
tuWas(new Katze());  
tuWas(new Dackel());  
...
```

# Basisklasse Object

- **Klasse ohne explizite Basisklasse**

- erbt von `java.lang.Object`
  - alle Java-Klassen lassen sich implizit in Object umwandeln
- wichtige gemeinsame Funktionalität
  - **toString**: Zeichenfolgen-Darstellung für ein Objekt
    - in eigener Klasse überschreiben um passende Zeichenfolge für ein Objekt zu liefern
  - **getClass**: Objekt mit der Klasseninformation des aktuellen Objekts holen



# ***Basisklasse Object***

## ■ **Klasse Object**

- weitere gemeinsame Funktionalität
  - **equals**: Testen ob ein Objekt gleich einem anderen ist
    - muss beim Einsatz in manchen Collections überschrieben werden (s.u. Hash basierte Collections)
  - **hashCode**: einen Streuwert für ein Objekt berechnen
    - muss beim Einsatz in manchen Collections überschrieben werden (s.u. Hash basierte Collections)
  - **finalize**: externe Ressourcen freigeben
    - ist seit Java 9 obsolet, wurde durch die Interfaces Closeable und AutoCloseable ersetzt (s.u. try-with-resources)
  - **wait/notify/notifyAll**:
    - Abfolge von Aktionen zwischen Threads synchronisieren (s.u. Nebenläufige Programmierung/wait and notify)

Achtung: == und != vergleichen die Referenzen

# Boxing und Unboxing

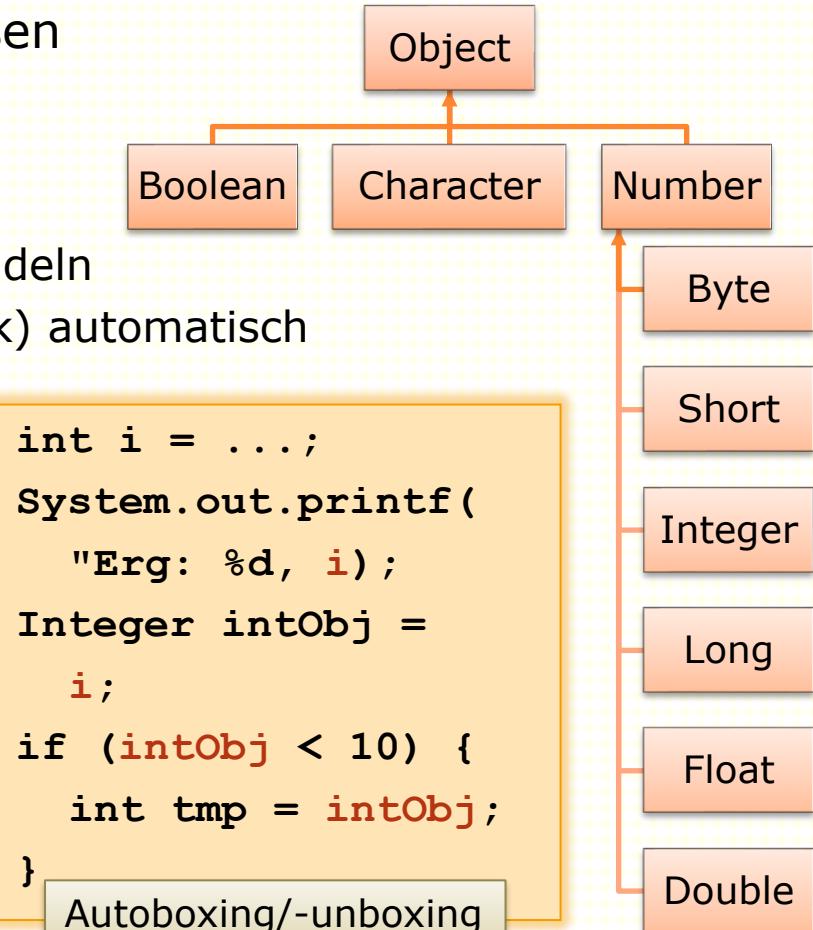
## Umwandlung Primitive Typen – Object

- Erfolgt über die Wrapperklassen

- Boxing: primitiven Wert in Wrapper-Objekt umwandeln
- Unboxing: Wrapper-Objekt in seinen primitiven Typ umwandeln
- Beides seit Java 5 (zum Glück) automatisch

```
int i = ...;
System.out.printf(
    "Erg: %d, Integer.valueOf(i));
Integer intObj =
    Integer.valueOf(i);
if (intObj.intValue() < 10) {
    int tmp = intObj.intValue();
}
```

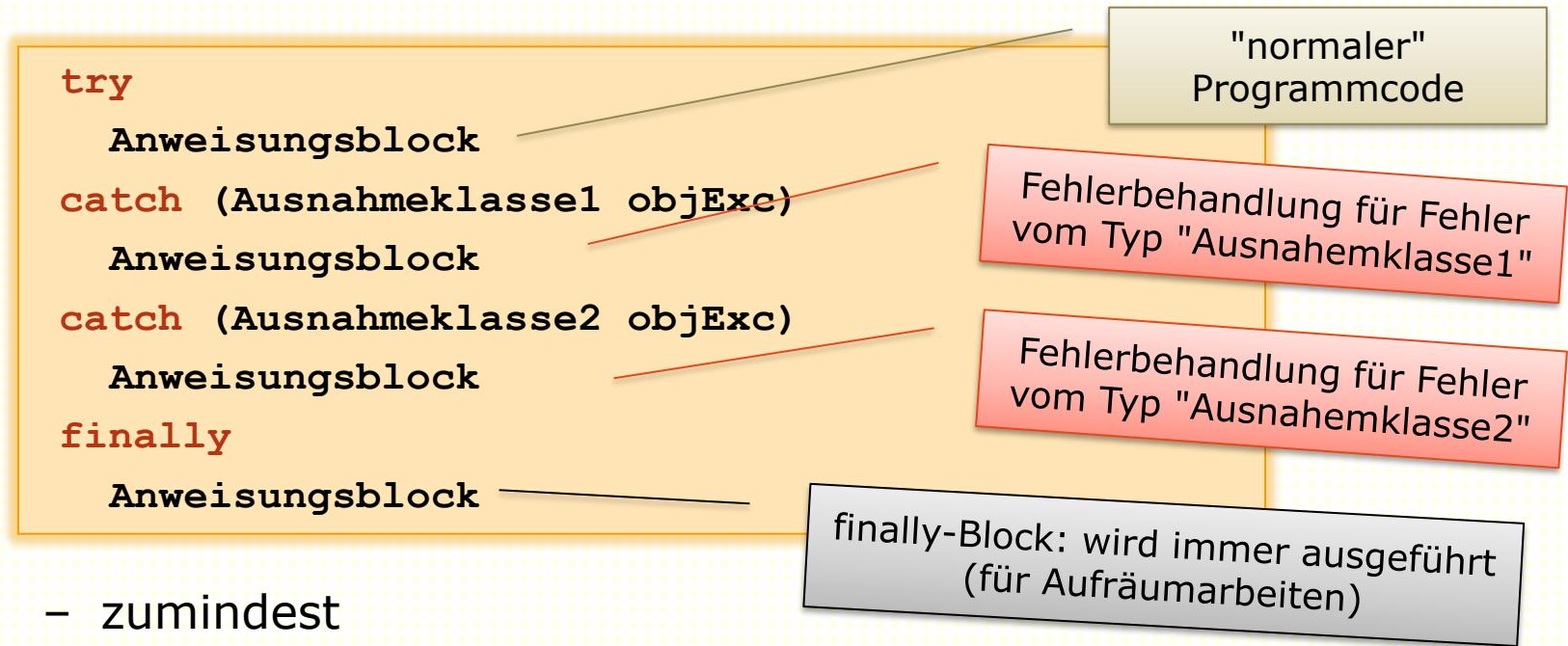
Explizites Boxing/Unboxing



# Fehlerbehandlung mit Exceptions

## ■ Fehlerbehandlung

- im Fehlerfall werden **Exceptions** ausgelöst, die mit **try-catch-finally**-Blöcken behandelt werden



- zumindest
  - try – 1 catch-Block oder
  - try – finally (der Fehler gilt dadurch nicht als behandelt)

# Fehlerbehandlung mit Exceptions

```
String strX = ..., strY = ...;  
try {  
    int x = Integer.parseInt(strX);  
    int y = Integer.parseInt(strY);  
    int erg = x / y;  
    System.out.println("Alles OK, Ergebnis = " + erg);  
} catch (ArithmeticalException e) {  
    // Fehlerbehandlung für ArithmeticalException  
    System.out.println("Fehler: " + e.toString());  
} catch (NumberFormatException e) {  
    // Fehlerbehandlung für NumberFormatException  
    System.out.println("Fehlerhafte Eingabe!");  
} finally {  
    // Code der jedenfalls ausgeführt wird  
    ...  
}
```

"normaler"  
Programmcode

Fehlerbehandlung für  
ArithmeticalException

Fehlerbehandlung für  
NumberFormatException

finally-Block: wird immer  
ausgeführt, falls vorhanden

# Fehlerbehandlung mit Exceptions

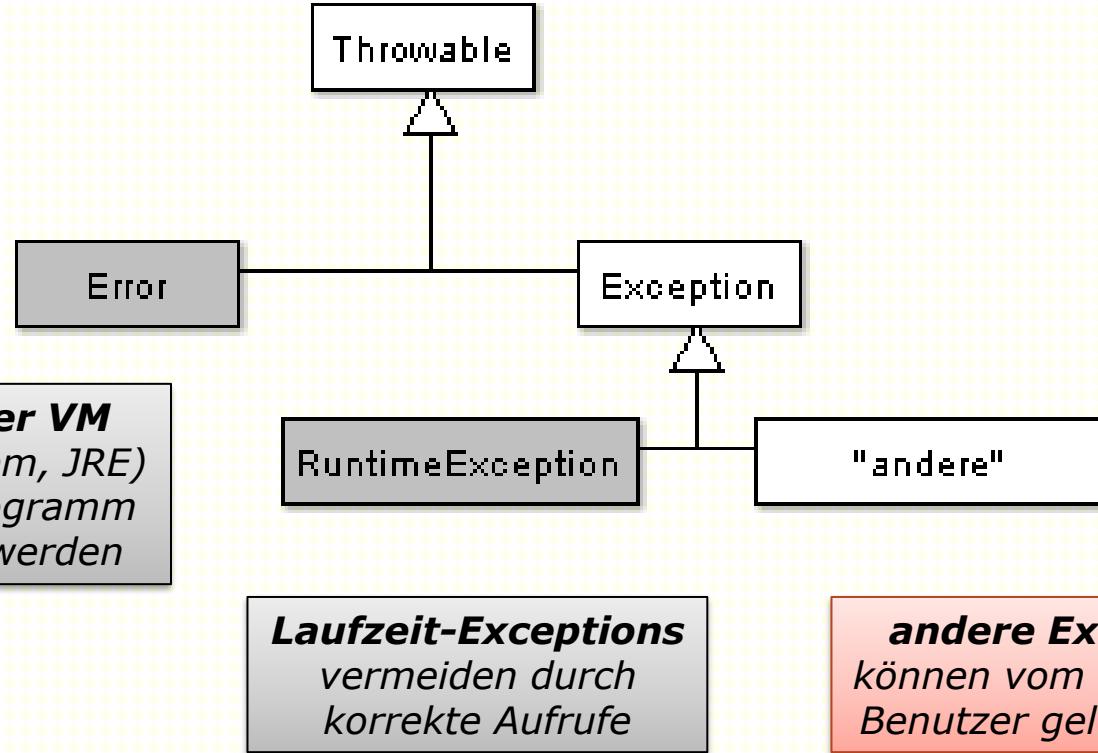
## ▪ Fehler auslösen

```
public static int calculate(char op, int z1, int z2) {  
    int erg;  
    // Berechnung je nach Operator  
    switch (op) {  
        case '+': erg = z1 + z2; break;  
        case '-': erg = z1 - z2; break;  
        case '/': erg = z1 / z2; break;  
        case '*': erg = z1 * z2; break;  
        default:  
            throw new IllegalArgumentException  
                ("Ungültiger Operator: " + op);  
    }  
    return erg;  
}
```

Die Ausführung wird abgebrochen und geht bei einem passenden catch-Block weiter

# Fehlerbehandlung mit Exceptions

## ▪ Vererbungshierarchie



# *Fehlerbehandlung mit Exceptions*

- **Unchecked Exceptions**
  - Error und RuntimeException
  - Müssen nicht abgefangen werden (kein try/catch)
- **Checked Exceptions (Catch or Specify)**
  - alle anderen
  - müssen behandelt werden
    - passender Catch-Block oder
    - Weiterreichen mit throws-Deklaration bei der Methoden-Signatur
  - andernfalls gibt es einen Compiler Fehler

# Fehlerbehandlung mit Exceptions

## ▪ Exception-Klassen selbst definieren

```
public class CalculationException extends Exception {  
    public CalculationException (String msg) {  
        super(msg);  
    }  
    public int calculate(char op, int z1, int z2)  
        throws CalculationException {  
        switch (op) {  
            case '+': return z1 + z2;  
            case '-': return z1 - z2;  
            case '/': return z1 / z2;  
            case '*': return z1 * z2;  
            default: throw new CalculationException (  
                "Unbekannter Operator " + op);  
        }  
    }  
}
```

eigene Klasse als Checked Exception

die Exception mit throws deklarieren

die deklarierte Exception werfen

# Fehlerbehandlung mit Exceptions

```
String strX = ..., strY = ...;  
try {  
    int x = Integer.parseInt(strX);  
    int y = Integer.parseInt(strY);  
    int erg = calculate('/', x, y);  
    System.out.println("Alles OK, Ergebnis = " + erg);  
} catch (CalculationException e) {  
    // Fehlerbehandlung für eigene Exceptionklasse  
    System.out.println("Fehler: " + e.getMessage());  
} catch (ArithmetricException e) {  
    // Fehlerbehandlung für ArithmetricException  
    System.out.println("Fehler bei einer Berechnung!");  
} catch (NumberFormatException e) {  
    // Fehlerbehandlung für NumberFormatException  
    System.out.println("Fehlerhafte Eingabe!");  
}
```

Aufruf einer Methode mit checked Exception

catch-Block für CalculationException ist erforderlich

# ***Java Programmierung***

Weiterführende Themen

# Nested classes

- **Statische eingebettete Klassen**
  - kann static Methoden und Variablen der umschließenden Klasse referenzieren
  - Verwenden (Instanziieren) unter Angabe der umschließenden Klasse
  - sonstiges Verhalten wie eine top-level Klasse

```
public class OuterClass{  
    public static class StaticNestedClass {  
        ...  
    }  
}
```

```
OuterClass.StaticNestedClass staticNested;  
staticNested = new OuterClass.StaticNestedClass();
```

# Nested classes

- **Innere eingebettete Klassen**

- ist mit einer Instanz der umschließenden Klasse verknüpft
- kann direkt auf die Instanzmethoden und -variablen dieses Objekts zugreifen
- Instanziierung über das umschließende Objekt

```
public class OuterClass{  
    private String name; <  
    public class InnerNestedClass {  
        public void print(){  
            System.out.println(OuterClass.this.name);  
        }  
    }  
}
```

Zugriff auf Instanzmember  
des "outer this"

```
OuterClass outerObject = new OuterClass();  
OuterClass.InnerNestedClass innerObject;  
innerObject = outerObject.new InnerNestedClass();
```

# Generics

- **Generische (=allgemeine) Typen und Methoden**
  - Definition von gleich bleibendem Verhalten für unterschiedliche Typen
  - für Klassen, Interfaces und Methoden

```
public class Box<T> {  
    private T value;  
    public T get () {  
        return value;  
    }  
    public void set (T value) {  
        this.value = value;  
    }  
}
```

```
Box<String> stringBox = new Box<String>();  
stringBox.set("123456");  
String strValue = stringBox.get();  
...
```

```
Box<Integer> intBox =  
    new Box<Integer>();  
intBox.set(10);  
int intValue = intBox.get();  
...
```

Als Typargument sind nur Klassen und Interfaces erlaubt -> statt primitivem Typ Wrapperklasse verwenden

- **Type erasure (Typlösung)**
  - Umsetzung in Java erfolgt mit Typlösung: Typen werden durch Object ersetzt
    - Zur Übersetzungszeit kann der Compiler die korrekte Verwendung erzwingen
    - Zur Laufzeit werden (implizite und explizite) Typumwandlungen durchgeführt
  - Probleme
    - raw type Deklaration (s. nächste Folie)
      - Fehler durch fehlerhafte Typumwandlungen
    - Explizite Umwandlung in generischen Typ
      - Typargument kann nicht verifiziert werden

## ▪ raw type Deklaration

- Deklarieren von generischen Typen ohne Typargumente

```
// problematisch
Box box = intBox;
// erlaubt aber falsch
box.setValue("123");
// ClassCastException!
int val = intBox.getValue();
```

Raw Type Deklaration kann zu **ClassCastException** führen!

## ▪ Platzhalter Deklaration

- Ermöglicht Zuweisung an allgemeineren Typen
- Verhindert Aufruf von Methoden, die das Typargument als Parameter verwenden

```
Box<?> box = intBox;
// erlaubt, liefert aber Object
Object val = box.getValue();
// nicht erlaubt
box.setValue(123);
box.setValue("123");
```

## ■ Bounds

- Einschränkungen bezüglich der als Typargument verwendbaren Typen
- garantieren, dass der verwendete Typ gewisse Operationen unterstützt (implementiert)

```
public class Box<T extends Comparable<T>> {  
    private T value;  
    ...  
    public boolean isGreater(T other) {  
        return value.compareTo(other) > 0;  
    }  
}
```

```
public class Person {  
    ...  
}
```

```
Box<Integer> iBox; // OK  
Box<LocalDate> dBox; // OK  
Box<String> sBox; // OK  
Box<Person> pBox; // Compiler Fehler
```

- **Generische Methoden**

- analog Klassen
- Typparameter können auch mit Bounds beschränkt werden

```
public class Utils{  
    public static <T extends Comparable<T>> T Max (T a, T b) {  
        T ret = a.compareTo(b) > 0 ? a : b;  
        return ret;  
    }  
}
```

```
int m1 = Utils.<Integer>Max(10, 20);  
// oder mit Typinferenz  
int m2 = Utils.Max(10, 20);
```

# Standardinterfaces: Sortierreihenfolge

- **Comparable<T>**

- Vergleich einer Instanz mit einem 2. Objekt
- Methode **compareTo(T o2)**
  - vergleicht die Instanz mit dem Objekt o2
- Definiert die "Natürliche Sortierreihenfolge"
  - wird beim Sortieren für den Default-Vergleich verwendet

```
class Employee implements Comparable<Employee> {  
    public int compareTo(Employee o2) {  
        if (o2 == null)  
            return 1;  
        int ret = 0;  
        ... // Vergleichen  
        return ret;  
    }  
}
```

Ergebnis des Vergleichs	
negativ	Instanz <b>kleiner</b> als o2
0	Instanz und o2 sind <b>gleich</b>
positiv	Instanz <b>größer</b> als o2

# *Standardinterfaces: Sortierreihenfolge*

- **Comparator<T>**

- Vergleich von zwei Objekten
- Methode **compare(T o1, T o2)**
  - vergleicht die beiden Objekte o1 und o2
- kann über overloads beim Sortieren übergeben werden

Ergebnis des Vergleichs	
negativ	o1 <b>kleiner</b> als o2
0	o1 und o2 sind <b>gleich</b>
positiv	o1 <b>größer</b> als o2

- **bis Java 7**

- Date
  - kapselt einen UTC-Zeitpunkt als Millisekunden seit dem 1.1.1970 ("epoch time")
  - sollte nicht für Berechnungen verwendet werden
- Calendar
  - Zeitpunkt in einem bestimmten Kalendersystem, z.B.
    - GregorianCalendar – Zeitpunkt im gregorianischen Kalender
  - Methoden für Datumsmanipulation (z.B. Zeitspanne zu Datum hinzufügen)
  - mühsam zu verwenden (Monatsnummer gehen von 0-11)
- DateFormat
  - Für Umwandlung von / nach String

# Date und Calendar

```
// neues Calendar-Objekt holen (bei uns GregorianCalendar)
Calendar cal = Calendar.getInstance();
// 1 Monat dazu
cal.add(Calendar.MONTH, 1);
// Umwandeln nach Date und long
Date dateToStore = cal.getTime();
long msToStore = cal.getTimeInMillis();

// Neues Date-Objekt erzeugen
Date aDate = new Date();
// Formatierungs-Objekt holen
DateFormat fmt = DateFormat.getDateInstance();
// Datum in Zeichenfolge umwandeln
String strDate = fmt.format(dateToStore);
```

- **Seit Java 8 neue Time API**
  - konsistente Implementierung, aber sehr umfangreich
  - **Instant**
    - kapselt einen UTC-Zeitpunkt ("epoch time")
      - wenn möglich auf Nanosekunden genau
      - sonst mindestens in Millisekunden
    - ersetzt Date
    - unterstützt keine Berechnungen
  - diverse Klassen für Datums- und Zeitwerte
    - mit oder ohne Zeitzone
    - unterstützen Berechnungen
  - Formatierung und Parsen
    - wird von vielen Klassen direkt unterstützt
    - DateTimeFormatter: mehr Kontrolle über Pattern

# Java Time API

Klasse	Bedeutung	Beispiel
Instant	UTC Zeitpunkt	Zeitmessung
LocalDate	Datum ohne Zeitzone	Geburtsdatum
LocalTime	Uhrzeit ohne Zeitzone	Öffnungszeiten
LocalDateTime	Datum+Uhrzeit, ohne Zeitzone	Beginn eines Seminars
ZonedDateTime	Datum+Uhrzeit, mit Zeitzone	Beginn eines Online-Kurses
ZoneId	Eine der vordefinierten Zeitzonen	Vordefinierte Zeitzone
Period	Intervall zwischen zwei Datumswerten, ohne Uhrzeit	Alter einer Person berechnen
Duration	Intervall zwischen zwei Datumswerten, mit Uhrzeit	Zeitmessung

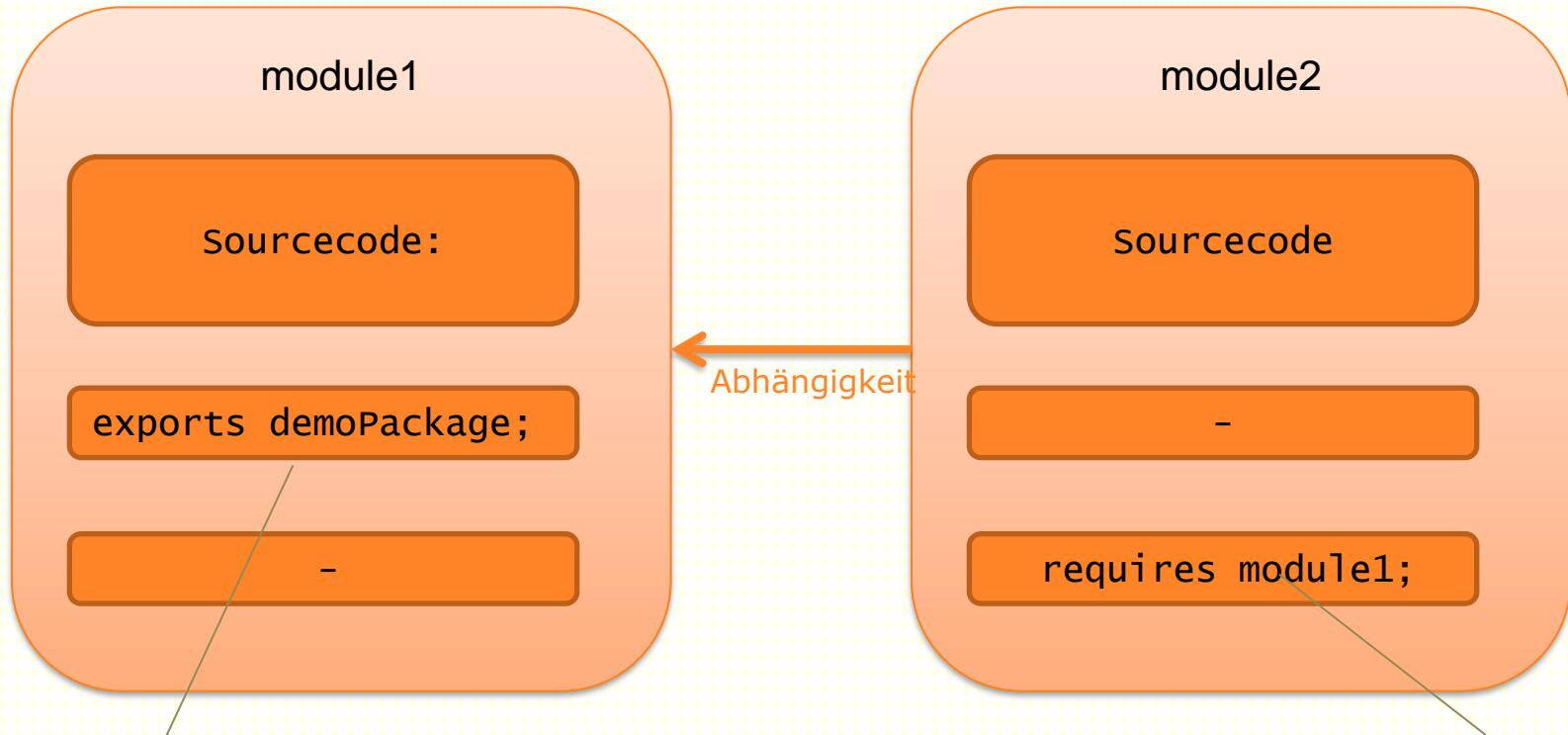
- **Immutable Objects**
    - alle Klassen erzeugen immutable objects
    - keine unabsichtliche Änderung von Werten
  - **Factory Methoden**
    - heißen in allen Klassen gleich:
      - of erzeugt ein Objekt aus mehreren Teil-Werten
      - from wandelt ein Objekt in einen anderen Typ um
      - with wendet eine Berechnung auf ein Objekt an
  - **Verkettete Calls**
    - möglich, weil die meisten Methoden
      - nicht void sind
      - nicht null zurückliefern
- ```
LocalDate dt = LocalDate.now()  
    .with(ChronoField.DAY_OF_MONTH, 15)  
    .minus(2, ChronoUnit.DAYS)
```

# **JAVA 9**

## ***Modulsystem***

# Java 9 Modulsystem

- **Abhängigkeiten zwischen Modulen**



Außerhalb des Moduls  
verwendbare Klassen werden  
über ihre Packages exportiert

Die Abhängigkeit wird über das Modul definiert.  
module2 kann alle Packages verwenden, die  
von module1 exportiert wurden

# Java 9 Modulsystem

- **Modulare Anwendung/Library**
  - definiert das module im Root-Verzeichnis der \*.class-Files
  - im File `module-info.java`
- **Neue Schlüsselwörter**
  - `module`: definiert ein Modul
  - `exports`: exportiert ein Package
  - `requires`: deklariert Abhängigkeit zu anderem Modul
- **Modulnamen und exportierte Packagenamen**
  - müssen im Ausführungskontext einer JVM eindeutig sein

```
module module1 {  
    exports demoPackage;  
}
```

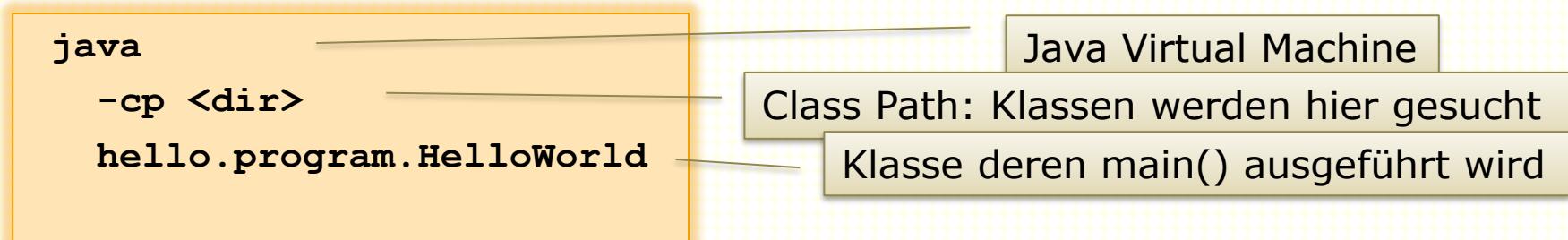
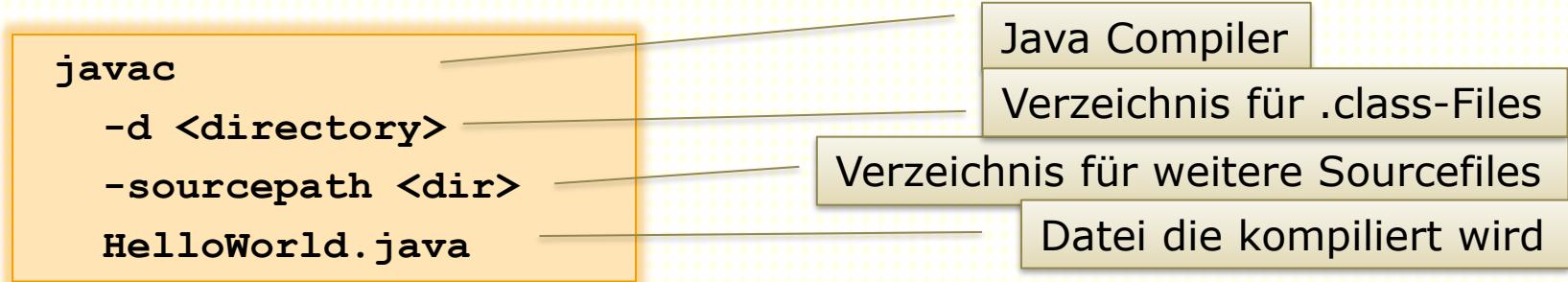
```
module module2 {  
    requires module1;  
}
```

# **Module Path vs. Class Path**

- **Module Path**
  - Dient zur Auflösung der Typen im Modulsystem
    - Reflection ist nur eingeschränkt möglich
- **Class Path**
  - Kann zusätzlich angegeben werden
  - ist für die Auflösung von nicht modularen Libraries erforderlich
  - Wenn nur der Class Path angegeben wird
    - läuft die Anwendung komplett im Kompatibilitätsmodus
- **Modulare Anwendung/Library im Class Path**
  - laufen wie nicht-modulare:
    - Alle Typen sind public
    - Reflection ist ohne Einschränkung möglich

# Kompilierung & Ausführung: WH

## ▪ Java Compiler und Java Runtime Environment



**Java Virtual Machine (JVM, VM) = Java Runtime Environment (JRE)**

```
C:\Test>javac -d bin -sourcepath src src\hello\program\Helloworld.java  
C:\Test>java -cp bin hello.program.Helloworld
```

# Kompilierung & Ausführung modular

## ▪ Commandline Argumente

```
javac
```

```
...
```

```
-cp <classpath>
```

```
-p <modulepath>
```

```
*.java
```

Pfad zu weiteren Libraries

Pfad zu modularen Libraries

Dateien die kompiliert werden

```
java
```

```
...
```

```
-p <modulepath>
```

```
-m myModule/prog.Program
```

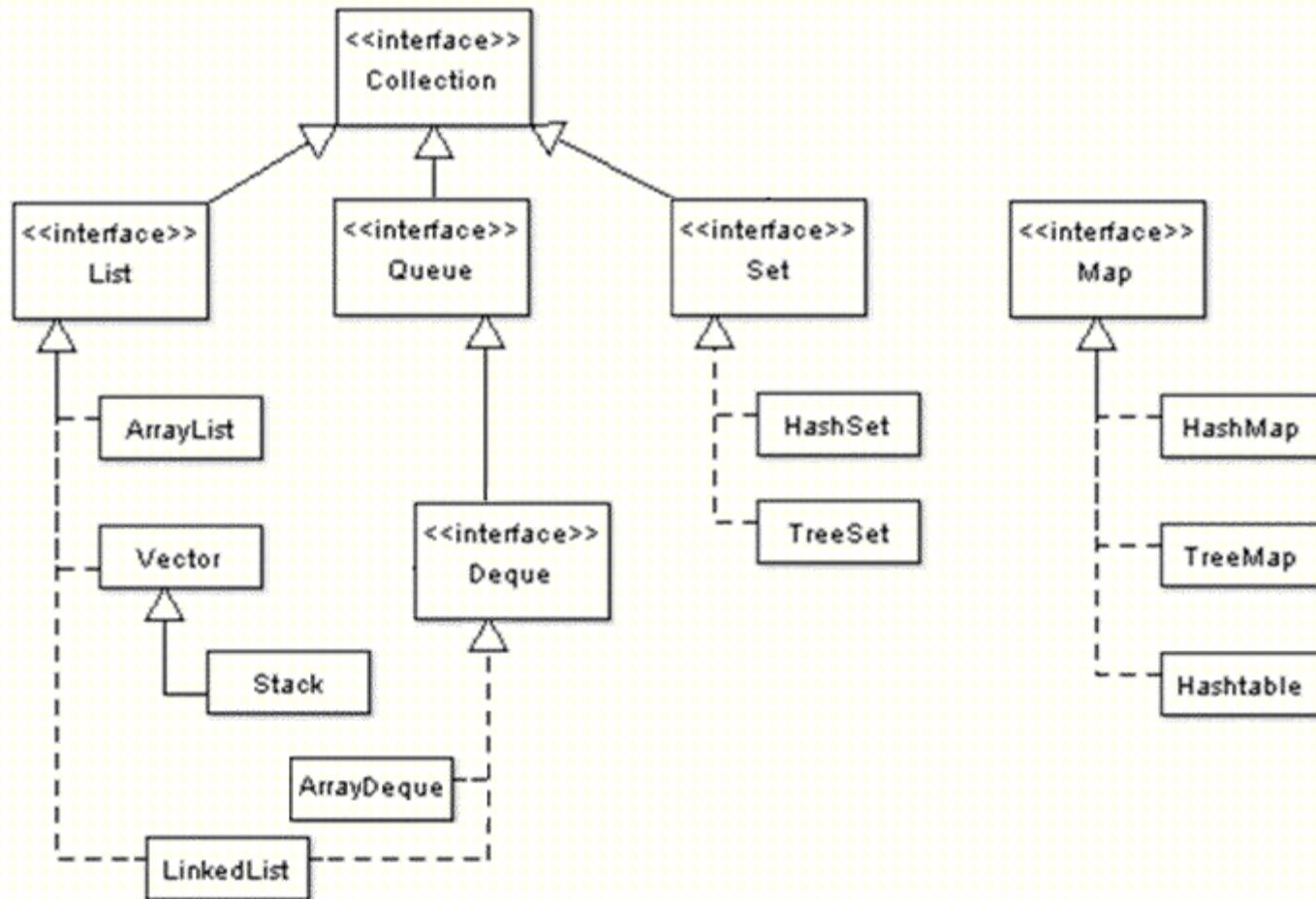
Pfad zu den Modulen

Modul/Klasse mit der main() Methode

```
javac -p build\lib -d build\app -sourcepath App\src App\src\prog\Programm.java  
java -p build\app;build\lib -m myModule/prog.Program  
java -cp build\app;build\lib prog.Program
```

# ***JAVA Collection Framework***

# Collections Übersicht



# *Standardschnittstellen*

- **Interface Collection<E>**
  - Basisinterface für Iterierbare Collections
  - add, contains, clear, remove, size, iterator
  - Verwendbar in for-each Schleifen
- **Interface Iterator<E>**
  - boolean hasNext()
    - ob es ein nächstes Element gibt
  - E next()
    - liefert das nächste Element und positioniert um 1 weiter
  - remove()
    - entfernt das zuletzt mit next gelieferte Element

# Collections iterieren

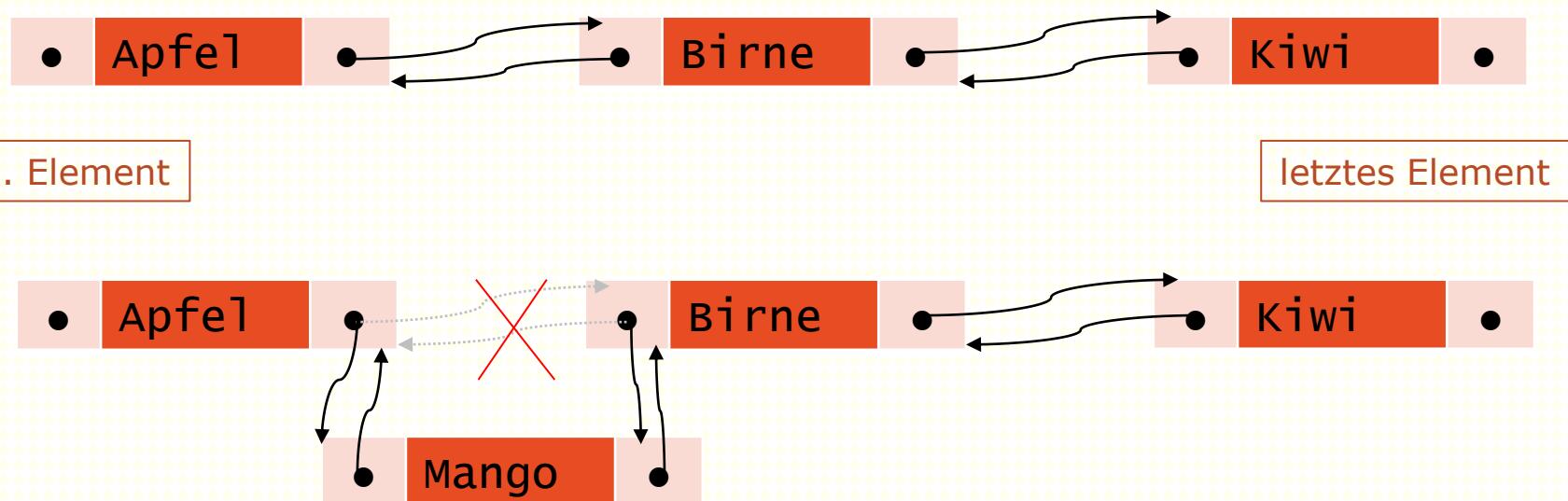
```
Collection<String> elems = ...;  
Iterator<String> iterator = elems.iterator();  
while(iterator.hasNext()) {  
    String s = iterator.next();  
    System.out.println(s);  
    if(s.equals("Bob"))  
        iterator.remove();  
}
```

```
// Alternativ mit for-each  
Collection<String> elems = ...;  
...  
for(String s: elems){  
    System.out.println(s);  
}
```

- **List**
  - Basisinterface für **Index basierte** Listen
  - Elemente sind **geordnet** nach der Indexposition
  - add, remove, size, indexof, lastIndexof, get
- **Implementierungen**
  - **ArrayList**:
    - verwaltet die Elemente mit einem **Array**
    - wird bei Bedarf automatisch neu allokiert
  - **vector**:
    - wie ArrayList, mit **synchronized** methods
  - **LinkedList**
    - verwaltet die Elemente mit einer **doppelt verketteten** Liste
    - implementiert auch das Interface **Deque**

# Doppelt verkettete Liste

- **Verwaltet die Elemente mit verketteten Knoten**
  - jeder Knoten hat potenziell 2 Nachbarn:
    - einen Vorgänger
    - einen Nachfolger
  - rasche Einfüge- und Löschoperationen



- **Klasse Stack**
  - von `vector` abgeleitete Stack-Implementierung
  - Elemente sind **geordnet** nach der umgekehrten Einfügereihenfolge (LIFO)
  - zusätzliche Methoden
    - `push`, `peek`, `pop`, `empty`
  - **Alternative: Deque-Implementierungen**
    - sollten bevorzugt für Stapel verwendet werden

# Warteschlange I

- **Queue**
  - Basisinterface für Warteschlangen (FIFO)
  - Elemente sind geordnet nach der Einfügereihenfolge
  - 2 Sets von Methoden
    - add, remove, element (werfen ggf. Exception)
    - offer, poll, peek (liefern false bzw. null)
- **Implementierungen**
  - ArrayDeque:
    - verwaltet die Elemente mit einem Array
    - wird bei Bedarf automatisch neu allokiert
  - LinkedList
    - verwaltet die Elemente mit einer doppelt verketteten Liste
    - implementiert auch das Interface Deque

# Warteschlange II

- **Deque (Double ended Queue)**
  - Basisinterface für **Warteschlangen** mit zwei Enden
  - Elemente sind nach ihrer Einfügereihenfolge und -position von vorne nach hinten **geordnet**
  - 2 Sets von Methoden für Operationen an beiden Enden
    - addFirst/addLast, removeFirst/removeLast, getFirst/getLast (werfen ggf. Exception)
    - offerFirst/offerLast, pollFirst/pollLast, peekFirst/peekLast (liefern false bzw. null)
- **Implementierungen**
  - ArrayDeque, LinkedList (siehe interface Queue)
- **Alternative für Stack**
  - addFirst/removeFirst/peekFirst ODER
  - addLast/removeLast/peekLast

# Wertemengen

- **Set**
  - Basisinterface für Mengen von **eindeutigen** Werten
  - add, remove, contains, size
- **Implementierungen**
  - HashSet
    - verwaltet die Elemente nach ihrem Hashcode in Hashtabellen
    - ungeordnet
  - TreeSet
    - verwaltet die Elemente in einem binären Suchbaum
    - sortiert nach den Werten

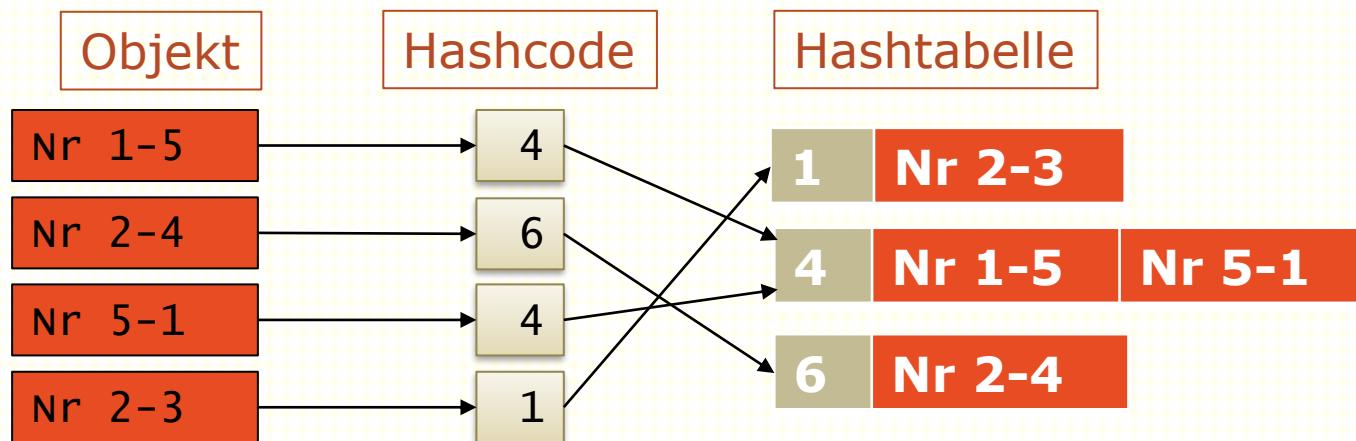
# Zuordnungen

- **Map**
  - Basisinterface für **Key-Value-Collections** mit **eindeutigen Keys**
  - `put`, `get`, `remove`, `size`, `containsKey`,  
`containsValue`, `entrySet`, `keySet`, `values`
- **Implementierungen**
  - `HashMap`
    - verwaltet die Paare nach dem Hashcode der Key-Elemente in **Hashtabellen**
    - **ungeordnet**
  - `TreeMap`
    - verwaltet die Paare in einem **binären Suchbaum**
    - **sortiert** nach den Key-Elementen
  - `Hashtable`
    - wie `HashMap`, mit **synchronized** methods

# Hash basierte Collections

- **Hashtabellen für die Ablage der Elemente**
  - Hashcode eines Elements bestimmt, in welchem Behälter das Element abgelegt wird

```
public class PersonalNr {  
    private int abteilung, nummer;  
    @Override  
    public String toString() {  
        return abteilung  
            + "-" + nummer;  
    }  
    ...  
}
```



# Hash basierte Collections

- **Hashcode eines Elements**
  - muss korrekt sein, damit Elemente **nach ihrem Wert aufgefunden** werden
  - `int hashCode()`
    - liefert den Hashcode für ein Objekt (Default: meistens die Speicheradresse des Objekts)
    - Korrekte Implementierung
      - **muss** für **gleiche Objekte** den **gleichen Wert** liefern
      - **kann** für **unterschiedliche Objekte** den **gleichen Wert** liefern
- **Wertevergleich**
  - `boolean equals(Object o2)`
    - führt den **Wertevergleich** des aktuellen Objekts mit dem Objekt `o2` durch
    - Default-Implementierung: Vergleich der Referenzen

# Hash basierte Collections

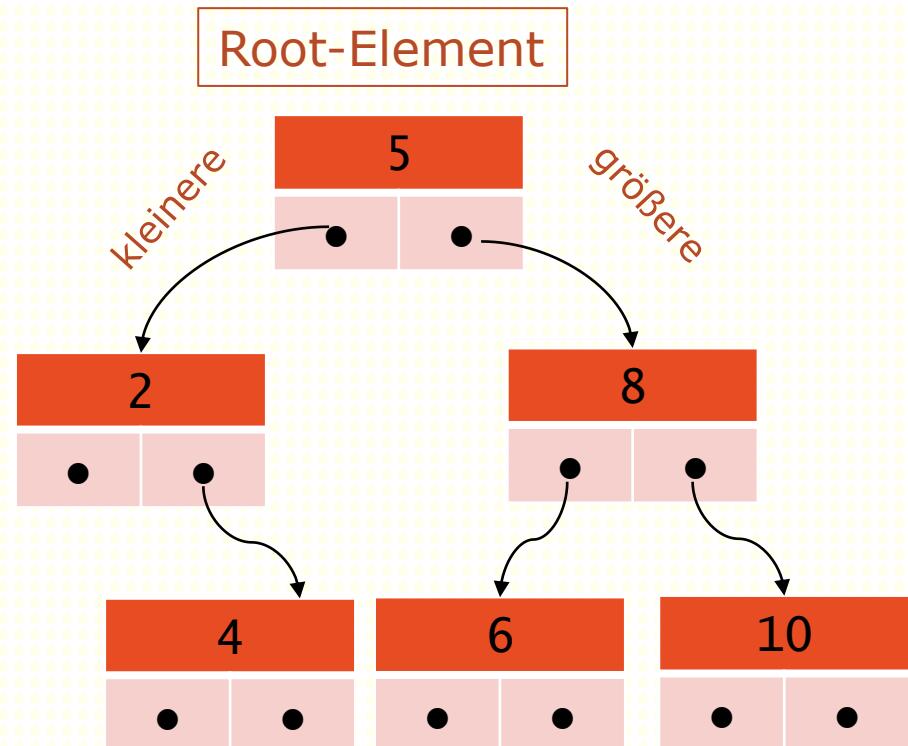
```
public class PersonalNr {  
    private int abteilung, nummer;  
  
    @Override public int hashCode() {  
        return abteilung ^ nummer;  
    }  
  
    @Override public boolean equals(Object o) {  
        if (!(o instanceof PersonalNr))  
            return false;  
        PersonalNr pnr = (PersonalNr) o;  
        return abteilung == pnr.abteilung  
            && nummer == pnr.nummer;  
    }  
}
```

HashCode für das Objekt liefern

detaillierten Wertevergleich durchführen

# Tree-Collections

- **Binärer Suchbaum für die Ablage der Elemente**
  - von einem Root-Element ausgehend hat jedes Element potenziell 2 Nachfolger:
    - auf der einen Seite einen kleineren Wert
    - auf der anderen Seite einen größeren Wert
  - ist automatisch sortiert



# Tree-Collections

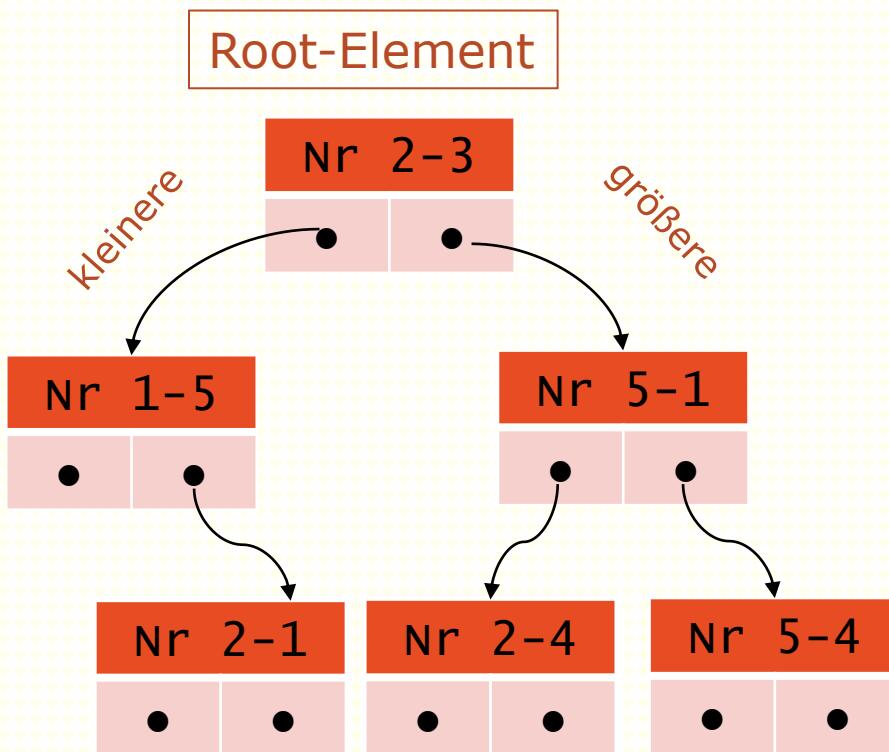
- **Elemente müssen sortierbar sein**
  - über die Comparable<E>-Implementierung des Element-Typs
    - natürliche Sortierreihenfolge
  - oder mit einem eigenen Comparator<E>
    - wird bei der Erzeugung angegeben

```
// eine Menge von Strings, mit Unterscheidung von
// Groß/Kleinschreibung
Set<String> fruits1 = new TreeSet<>();

// eine Menge von Strings, ohne Unterscheidung von
// Groß/Kleinschreibung
Set<String> fruits2
    = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
```

# Tree-Collections

- Elemente können auch eigene Typen sein



```
class PersonalNr implements Comparable<PersonalNr> {  
    private int abteilung, nummer;  
  
    @Override public int compareTo(PersonalNr o) {  
        int cmp = 0;  
        // zuerst nach abteilung,  
        // dann nach nummer sortieren  
        ...  
        return cmp;  
    }  
}
```

# Eigenschaften von Collections

|            | List | Queue | Deque | Set     | Map     |
|------------|------|-------|-------|---------|---------|
| Iterator   | x    | x     | x     | x       |         |
| Index      | x    |       |       |         |         |
| Geordnet   | x    | x     | x     |         |         |
| Ungeordnet |      |       |       | HashSet | HashMap |
| Sortiert   |      |       |       | TreeSet | TreeMap |
| Eindeutig  |      |       |       | x       | x       |

x ... gilt für alle Implementierungen der Collection

Klasse ... gilt nur für diese Implementierung

# ***Funktionale Programmierung***

Functional Interfaces, Lambda Expressions,  
Methodenreferenzen und die Stream API

# Anonyme Interface Implementierung

## ▪ Implementierung

- erfolgt direkt an der Stelle an der das Objekt benötigt wird

```
public interface AnimalFilter {  
    boolean isTrueFor(Animal a);  
}
```

Instanziierung

Basis-Interface oder Klasse

```
AnimalFilter filter1 = new AnimalFilter()  
{  
    @Override  
    public boolean isTrueFor(Animal a) {  
        return a.isHerbivore();  
    }  
};
```

Implementierung der anonymen Klasse

; schließt die Deklarations-Anweisung ab

# *Functional Interfaces*

- **Interfaces mit nur 1 abstrakter Methode**
  - können als Functional Interfaces eingesetzt werden
  - optionale Kennzeichnung mit @FunctionalInterface Annotation
    - Compilerfehler wenn das Interface weitere abstrakte Methoden definiert
  - enthalten häufig static oder default Methoden
  - seit Java 8

```
@FunctionalInterface  
public interface AnimalFilter {  
    boolean isTrueFor(Animal a);  
}
```

# Lambda-Ausdrücke

## ▪ Lambda Expressions

- kompakte Syntax für die Implementierung eines Functional Interface
- Alternative Syntax um ein Interface anonym zu implementieren

## ▪ Syntax

- lässt alles weg, was der Compiler aus dem Kontext ermitteln kann:
  - die Namen von Interface und Methode entfallen
  - Typen der Parameter können vom Compiler ermittelt werden
  - Returntyp wird immer vom Compiler ermittelt

```
(argument list) -> expression or code block
```

# Lambda-Ausdrücke

```
AnimalFilter filter1 = new AnimalFilter() {  
    @Override public boolean isTrueFor(Animal a) {  
        return !a.isHerbivore();  
    }  
};
```

Anonyme Klasse

Basis-Interface

```
AnimalFilter filter2 = (a) -> {  
    return !a.isHerbivore();  
};
```

Implementierung der  
abstrakten Methode

Lambda Expression

; schließt die Deklaration ab

```
AnimalFilter filter2 = a -> !a.isHerbivore();
```

Bei einzelner Anweisung dürfen Blockklammern und return entfallen

# Methodenreferenzen

- **Method reference**
  - Alternative Syntax für Lambda Expressions
  - Interface-Implementierung durch Verweis auf passende Methode oder Konstruktor
- **Syntax**

```
<Methodenhalter>::<Methodename>
```

| Syntax                      | Kind                                              |
|-----------------------------|---------------------------------------------------|
| ClassName::staticMethodName | Referenz auf statische Methode                    |
| objName::methodName         | Referenz auf Instanzmethode                       |
| ClassName::methodName       | Referenz auf Instanzmethode mit arbiträrem Objekt |
| className::new              | Referenz auf einen Konstruktor                    |

# Methodenreferenzen

## ▪ Referenz auf statische Methode

```
public class AnimalUtil{  
    public static boolean isVegetarian(Animal a)  
    { return a.isHerbivore(); }  
}
```

statische  
Methode

```
AnimalFilter filter2 = a -> AnimalUtil.isVegetarian(a);
```

Lambda  
Expression

```
AnimalFilter filter2 = AnimalUtil::isVegetarian;
```

Method  
Reference

## ▪ Referenz auf Instanzmethode mit arbiträrem Objekt

```
AnimalFilter filter2 = a -> a.isHerbivore();
```

Lambda  
Expression

```
AnimalFilter filter2 = Animal::isHerbivore;
```

Method  
Reference

# Vordefinierte Functional Interfaces

| Interface     | Methode          | Beschreibung                                                          |
|---------------|------------------|-----------------------------------------------------------------------|
| Supplier<T>   | T get()          | Ein Element bereitstellen                                             |
| Consumer<T>   | void accept(T)   | Ein Element verarbeiten                                               |
| Predicate<T>  | boolean test(T)  | Eine Bedingung ("predicate") für ein Element prüfen                   |
| Function<T,R> | R apply (T)      | Eine Funktion auf ein Element anwenden und das Ergebnis zurückliefern |
| Comparator<T> | int compare(T,T) | Den Vergleichswert für zwei Objekte zurückliefern                     |

## ■ Weitere Interfaces

- für die Verarbeitung von 2 Elementen
  - BiConsumer, Bi...
- für die Verarbeitung von ausgewählten Primitiven
  - IntConsumer, LongConsumer, ...

# Vordefinierte Functional Interfaces

## ▪ Beispiel Predicate

```
Predicate<Animal> filter = a -> AnimalUtil.isVegetarian(a);
```

```
Predicate<Animal> filter = AnimalUtil::isVegetarian;
```

```
Predicate<Animal> filter =  
    Predicate.not(AnimalUtil::isVegetarian);
```

## ▪ Beispiel Comparator

```
Comparator<Animal> comparator =  
(a1, a2) -> a1.getWeight() - a2.getWeight();
```

```
Comparator<Animal> comparator =  
    Comparator.comparing(Animal::isHerbivore)  
        .thenComparing(Animal::getWeight);
```

- **Stream interface**

- gibt Zugriff auf eine Sequenz von Elementen
- mit Unterstützung für Filterung und Sortierung
- Pipeline Verarbeitung - Operationen werden verkettet

**Source → intermediate operation(s) → terminal operation**

- Intermediate Operations liefern den Stream zurück damit verkettete Calls möglich sind
- Für ausgewählte Grunddatentypen spezialisierte Interfaces
  - IntStream, LongStream, DoubleStream

```
List<Animal> animalList = ...;

animalList.stream() // Quelle
    .filter(Animal::isHerbivore) // Intermediate Operation
    .forEach(System.out::println); // Terminal Operation
```

## ■ Intermediate Operations

- **filter(Predicate<T>):**
  - Elemente gemäß dem Predicate filtern
- **sorted() / sorted(Comparator<T>):**
  - Sortieren in natürlicher Sortier-Reihenfolge bzw. gemäß dem Comparator
- **map(Function<T,R>):**
  - liefert Stream von Elementen, die mit der Function aus den Quell-Elementen berechnet werden
- **mapToIntToIntFunction<T>) /  
mapToLong(ToLongFunction<T>) /  
mapToDouble(ToDoubleFunction<T>):**
  - liefern Stream von primitiven Werten, die mit der Function aus den Quell-Elementen berechnet werden

## ▪ **Terminal Operations**

- **forEach(Consumer):**
  - führt Aktion für jedes Element aus
- **collect(Collector):**
  - liefert Collection mit den Elementen, vordefinierte Collectors sind z.B. `Collectors.toList()` oder `Collectors.toSet()`
- **findFirst():**
  - liefert das erste Element als `Optional<T>`
- **count():**
  - liefert Anzahl der Elemente
- **min(Comparator), max(Comparator):**
  - liefert das min/max Element (`Optional`)
- **sum(), average():**
  - liefert Summe/Durchschnitt (nur für primitive Streams)

# Optionale Ergebnisse

## ▪ Klasse Optional <T>

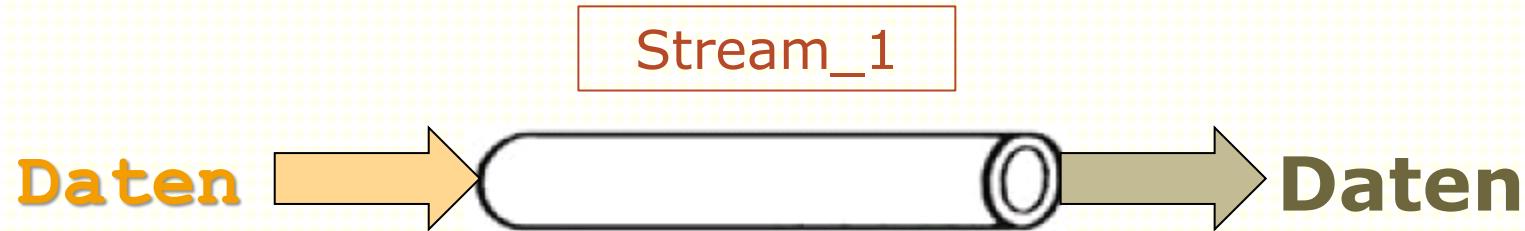
- Wrapper für einen Wert, der vorhanden sein kann oder auch nicht
  - vereinfacht das Handlen von null-References
  - wird von einigen Terminal Operations zurückgeliefert
- Methoden
  - **ifPresent**, **ifPresentOrElse**: einen Consumer ausführen, wenn ein Wert vorhanden ist
  - **isPresent**, **isEmpty**: Prüfen ob ein Wert vorhanden ist
  - **get**: den Wert holen
    - wirft eine NoSuchElementException, falls kein Wert vorhanden
  - **orElse**, **orElseGet**, **orElseThrow**: den Wert oder einen Alternativwert ermitteln bzw. eine Exception auslösen
  - **of**, **ofNullable**: ein Optional mit dem angegeben Wert erzeugen

```
animals.stream().findFirst()  
.ifPresent(System.out::println);
```

# ***Streams und FileIO***

# Streams

**Streams transportieren Daten von A nach B ...**



**... und können die Daten dabei bearbeiten**

**Streams können aneinander gereiht werden**

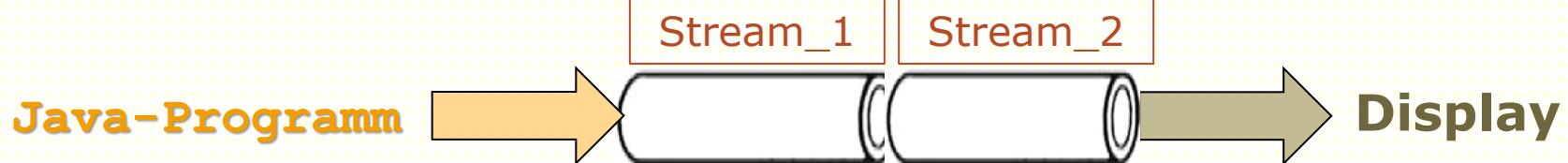


# Streams

**Das Ende der Leitung ist ein Java-Programm  
oder die Tastatur ...**

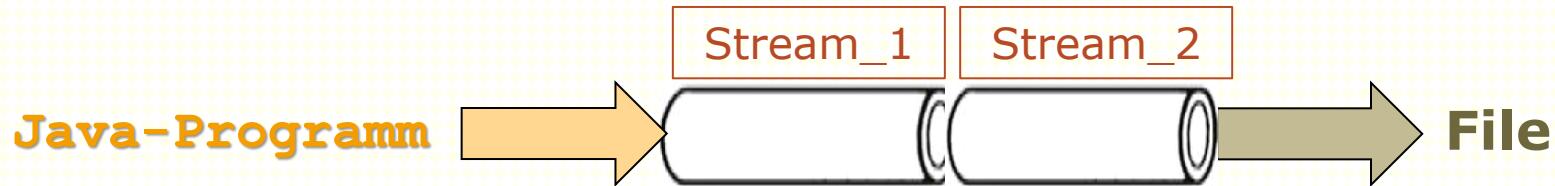


**... oder der Bildschirm ...**



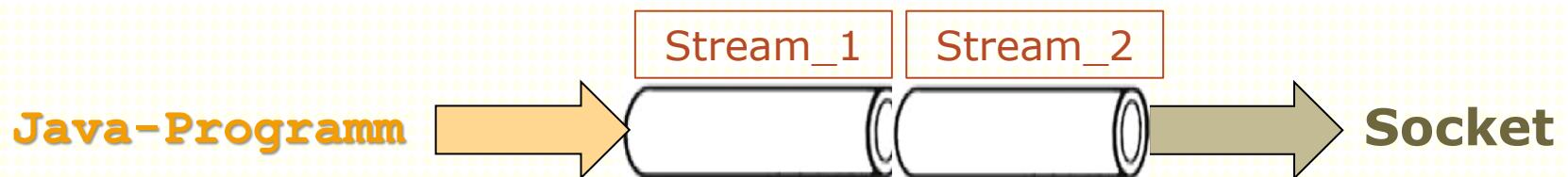
# Streams

... oder ein File ...



# Streams

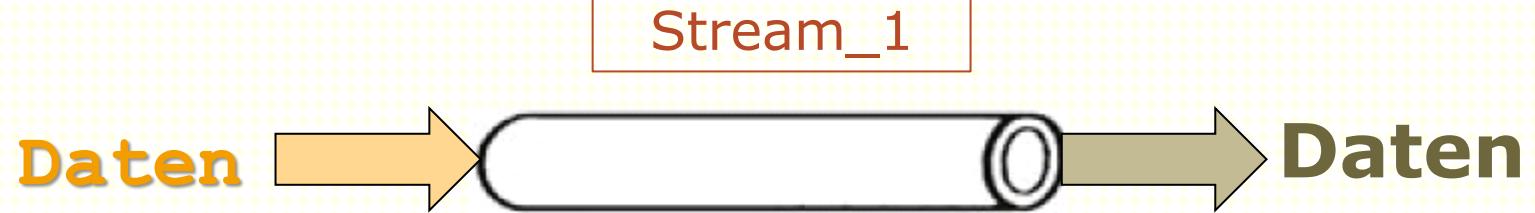
... oder das Ende einer Netzwerkverbindung  
(Socket)



oder ...

# Streams

**Streams sind immer one-way**



**... um ein File zu lesen UND zu schreiben braucht  
man 2 Streams**

# Arten von Streams

|                             | <b>Byte-Stream</b>                           | <b>Character-Stream</b>                                |
|-----------------------------|----------------------------------------------|--------------------------------------------------------|
| Zweck                       | Schreiben oder Lesen einer Sequenz von Bytes | Schreiben oder Lesen einer Sequenz von Unicode-Zeichen |
| Basisklasse für die Ausgabe | OutputStream                                 | Writer                                                 |
| Basisklasse für die Eingabe | InputStream                                  | Reader                                                 |
| Beispiel                    | FileOutputStream,<br>FileInputStream         | FileWriter<br>FileReader                               |

# Character-Streams

- **abstract class Writer**

- Basisklasse für Ausgabestreams, wichtige Methoden:

- `void write(int)`: ein Zeichen schreiben
    - `void write(char[])`: Zeichen blockweise schreiben
    - `void write(String)`: Zeichenfolge schreiben
    - `void flush()`: Flush durchführen (das bisher geschriebene ans Ziel schreiben und allfälligen Puffer leeren)

- **abstract class Reader**

- Basisklasse für Eingabestreams, wichtige Methoden:

- `int read()`: ein Zeichen lesen (Ergebnis ist das Zeichen)
    - `int read(char[])`: Zeichen blockweise in einen Puffer lesen, Ergebnis ist Anzahl der gelesenen Zeichen

# Beispiel *FileWriter* und -*Reader*

```
try {
    FileWriter fw = new FileWriter("test.txt");
    fw.write("Das ist eine Textdatei");
    fw.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Text an ein  
File schreiben

Text von  
einem File  
lesen

```
try {
    FileReader fr = new FileReader("test.txt");
    int x;
    while ((x = fr.read()) != -1) // -1 bedeutet EOF
        System.out.print((char) x);
    fr.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

# try-with-resources

- **Interfaces AutoCloseable, Closeable**
  - ermöglichen die Verwendung in einem try-with-resources-Statement
  - Schließen der Ressource erfolgt automatisch in einem impliziten finally Block
  - Ersatz für Überschreiben von Object.finalize

/ try-with-resources schließt den Stream automatisch

```
try (FileReader fr = new FileReader("test.txt")) {
    int x;
    while ((x = fr.read()) != -1) // -1 bedeutet EOF
        System.out.print((char) x);
    // fr.close(); // nicht erforderlich
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

# *Character-Streams*

| Klasse                   | Zweck                                    | Methoden                   |
|--------------------------|------------------------------------------|----------------------------|
| FileWriter<br>FileReader | Text schreiben bzw.<br>lesen an/von File |                            |
| PrintWriter              | Primitive Daten als<br>Text schreiben    | print<br>println<br>printf |
| BufferedWriter           | Zeilenweise<br>schreiben                 | newLine                    |
| BufferedReader           | Zeilenweise lesen                        | readLine                   |
| StringWriter             | An einen<br>StringBuffer<br>schreiben    | getBuffer<br>toString      |
| StringReader             | von einem String<br>lesen                |                            |

# Beispiel BufferedWriter- und Reader

```
try(BufferedWriter bw = new BufferedWriter(  
        new FileWriter("test.txt"))){  
    bw.write("Das ist eine Textdatei");  
    bw.newLine();  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

Zeilenweise  
Schreiben

```
try (BufferedReader br = new BufferedReader(  
        new FileReader("test.txt")){  
    String line;  
    while ((line = br.readLine()) != null) {  
        System.out.println(line);  
    }  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

Zeilenweise  
Lesen

- **abstract class OutputStream**
  - Basisklasse für Ausgabestreams, wichtige Methoden
    - `void write(int)`: ein Byte schreiben
    - `void write(byte[])`: Bytes blockweise schreiben
    - `void flush()`: Flush durchführen (das bisher geschriebene ans Ziel schreiben und allfälligen Puffer leeren)
- **abstract class InputStream**
  - Basisklasse für Eingabestreams, wichtige Methoden
    - `int read()`: ein Byte lesen (Ergebnis ist das Byte)
    - `int read(byte[])`: Byte blockweise in einen Puffer lesen, Ergebnis ist Anzahl der gelesenen Bytes
    - `byte[] readAllBytes()`: alle Bytes bis zum Ende des Streams in einen Puffer lesen, Ergebnis ist der Puffer

# Byte-Streams

| Klasse                               | Zweck                                     | Methoden                                                                                                                      |
|--------------------------------------|-------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| FileInputStream,<br>FileOutputStream | Schreiben bzw.<br>Lesen von / an<br>Datei |                                                                                                                               |
| DataOutputStream                     | Primitive Daten<br>binär schreiben        | <code>writeInt</code> , <code>writeChar</code> ,<br><code>writeBoolean</code> ,<br><code>writeUTF</code> ,                    |
| DataInputStream                      | Primitive Daten<br>binär lesen            | <code>readInt</code> , <code>readChar</code> ,<br><code>readBoolean</code> , <code>readUTF</code> ,<br><code>skipBytes</code> |
| ObjectOutputStream                   | Objekte schreiben<br>(Serialisierung)     | <code>writeObject</code>                                                                                                      |
| ObjectInputStream                    | Objekte lesen<br>(Serialisierung)         | <code>readObject</code>                                                                                                       |

# DataOutputStream - DataInputStream

```
try (DataOutputStream os = new DataOutputStream(  
    new FileOutputStream("daten.bin"))){  
    os.writeInt(10);  
    os.writeUTF("Hallo" );  
    os.writeChar('x');  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

Binärdaten  
schreiben

Binärdaten  
lesen

```
try (DataInputStream is = new DataInputStream(  
    new FileInputStream("daten.bin"))){  
    int x = is.readInt();  
    String s = is.readUTF();  
    char c = is.readChar();  
    System.out.printf("%d %s %c", x, s, c);  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

# *ObjectOutput- und ObjectInputStream*

## ■ **Serialisierung**

- Automatisiertes Schreiben und Lesen eines Objekt-Graphen
  - alle Instanzfelder (auch private) der Klasse werden in den Stream geschrieben bzw. vom Stream eingelesen
  - transient: kennzeichnet ein Feld das nicht serialisiert wird
- Der Graph muss komplett serialisierbar sein
- Unterstützt auch Arrays und Collections

## ■ **Interface Serializable**

- kennzeichnet eine Klasse als serialisierbar
- die Klasse sollte eine serialVersionUID haben:  
`private static final long serialVersionUID = 1L;`

- **Files sind auf allen Plattformen Byte-Orientiert**
  - Schreiben/Lesen von Unicode-Text erfordert Konvertierung zwischen Byte- und Character-Streams
  - erfolgt unter Verwendung eines Character Encodings, z.B.
    - UTF-7 , UTF-8 , UTF-16 (UCS Transformation Format)
    - US-ASCII (7-Bit ASCII)
    - ISO-8859-1 (ISO Latin Alphabet No. 1)
    - windows-1252 (ANSI, CP1252)
  - Klasse **Charset**
    - gibt Zugriff auf vordefinierte Character Encodings
    - Charset-Instanzen können über ihren Namen abgerufen werden, z.B. für UTF-8 Kodierung:  
`charset.forName("UTF-8")`

- **FileWriter und FileReader**
  - verwenden per Default das Encoding der Plattform
    - unter Windows (in Europa) meist CP1252
    - auf anderen Plattformen nicht einheitlich
    - kann mit VM-Argument geändert werden, z.B. auf UTF-8  
-Dfile.encoding=UTF-8
  - neue Konstruktoren (seit Java 11) erlauben Angabe eines Charset
- **Alternative: Brückenklassen direkt verwenden**
  - Kodierung kann im Konstruktor als String oder Charset angegeben werden
  - InputStreamReader
    - Unicode-Zeichen aus Bytesequenz lesen
  - OutputStreamWriter
    - Unicode-Zeichen in Bytesequenz schreiben

# Brückenklassen

```
try (Writer w = new FileWriter("test2.txt",
        Charset.forName("UTF-8"))){
    w.write("© by M&T");
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

UTF-8 File schreiben

UTF-8 File lesen

```
try (Reader r = new FileReader("test2.txt",
        Charset.forName("UTF-8")) {
    int c;
    while ((c = r.read()) != -1) {
        System.out.print((char) c);
    }
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

# Brückenklassen

```
try (Writer w = new OutputStreamWriter(  
        new FileOutputStream("test2.txt") , "UTF-8")) {  
    w.write("© by M&T");  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

UTF-8 File  
schreiben

UTF-8 File  
lesen

```
try (Reader r = new InputStreamReader(  
        new FileInputStream("test2.txt") , "UTF-8")) {  
    int c;  
    while ((c = r.read()) != -1) {  
        System.out.print((char) c);  
    }  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

# Hilfsklasse File

- **Hilfsklasse für Pfad- und Dateinamen**
  - Erzeugung mit absolutem oder relativem Pfad, mit oder ohne Parent-Verzeichnis
  - Zugriff auf Teile des Dateipfads
    - `getName`, `getParent`, `getAbsolutePath`, `getAbsoluteFile`
  - Prüfmethoden
    - `exists`, `isDirectory`, `isFile`, `length`
  - Files und Verzeichnisse verwalten
    - `createNewFile`, `mkdir`, `mkdirs`, `renameTo`, `delete`
    - `list`, `listFiles`
  - Zugriff auf temp. Dateien
    - `createTempFile`, `deleteOnExit`
  - Konstante für Pfad- und Verzeichnistrennzeichen
    - `pathSeparator`, `separator`

# Hilfsklassen Path, Paths und Files (nio)

- In **java.nio** wurde **Funktionalität von File geteilt**
  - Path repräsentiert einen File- oder Verzeichnisnamen
  - Paths: Hilfsklasse zum Verketten von Verzeichnisnamen
  - Files: Hilfsklasse zur Verwaltung von Files und Verzeichnissen
    - Erzeugen, Löschen, Kopieren, Verschieben
    - Öffnen von Files zum Lesen oder Schreiben
    - Zugriff mit Stream API auf Inhalt von Textfile bzw. Verzeichnis

```
try {  
    Path path = Paths.get("daten", "testdaten.csv");  
    Files.lines(path) // ein UTF-8 File zeilenweise lesen  
        .forEach(line -> System.out.println("Line: " + line));  
} catch (IOException e) { ... }
```

# XML Serialisierung

- **JAXB (Java Architecture for XML Binding)**
  - Mapping zwischen Instanz-Feldern bzw. -Properties eines Java-Objekts und XML-Elementen oder -Attributen
  - Annotationen passen die Mappings an
  - Klasse **JAXBContext**
    - Einstiegspunkt für Clients in die JAXB API
  - Interface **Marshaller**
    - Unterstützt die Serialisierung (marshal) an verschiedene Ziele
  - Interface **Unmarshaller**
    - Unterstützt die Deserialisierung (unmarshal) von verschiedenen Quellen
  - Ist eigentlich Teil der Java EE/Jakarta EE

Achtung: Seit Java 11 sind die JAX-APIs nicht mehr Teil der Java SE und müssen separat eingebunden werden

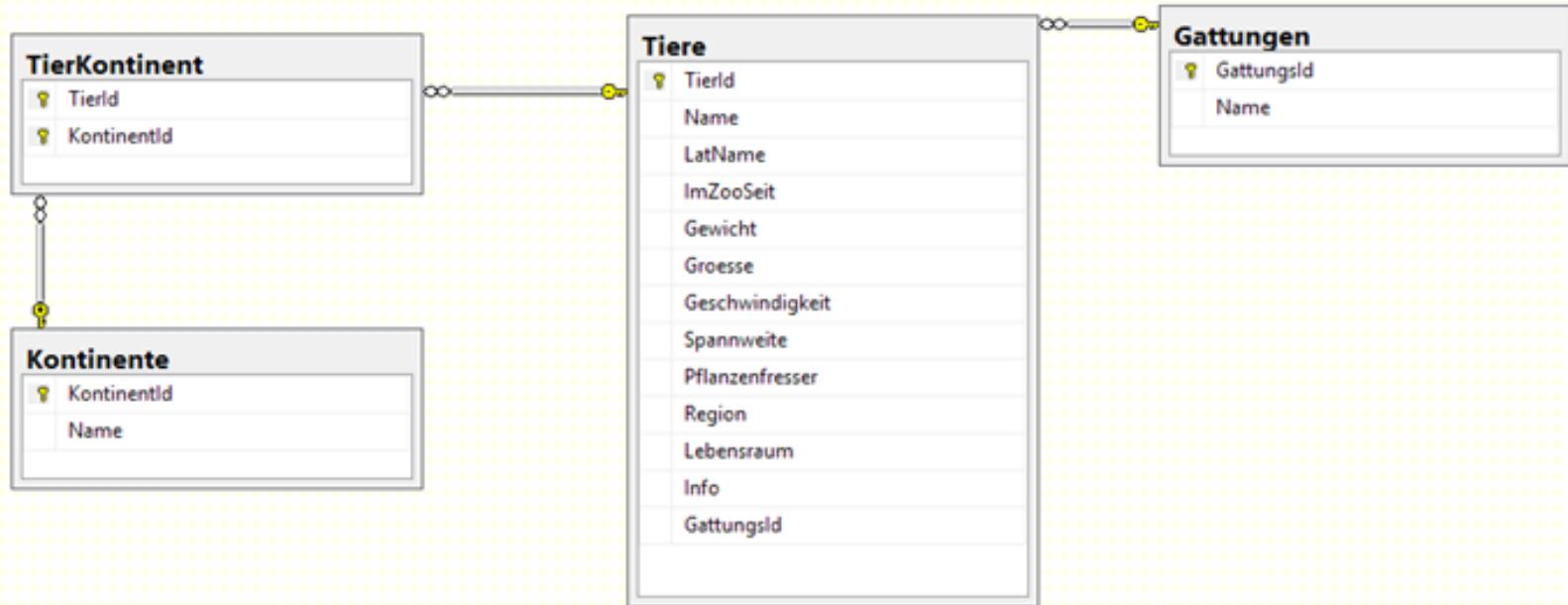
# **XML Serialisierung Annotationen**

| <b>Annotation</b> | <b>Bedeutung</b>                                             |
|-------------------|--------------------------------------------------------------|
| @XmlElement       | XML Element anpassen (Name, ob erforderlich)                 |
| @XmlAttribute     | die Eigenschaft als XML Attribut verarbeiten, Name anpassen, |
| @XmlTransient     | die Eigenschaft nicht verarbeiten                            |
| @XmlRootElement   | eine Klasse als Root-Element des XML-Dokuments zulassen      |
| @XmlAccessorType  | Art des Mappings steuern (über Felder oder über Properties)  |
| @XmlSeeAlso       | Abgeleitete Klassen einer Vererbungshierarchie registrieren  |

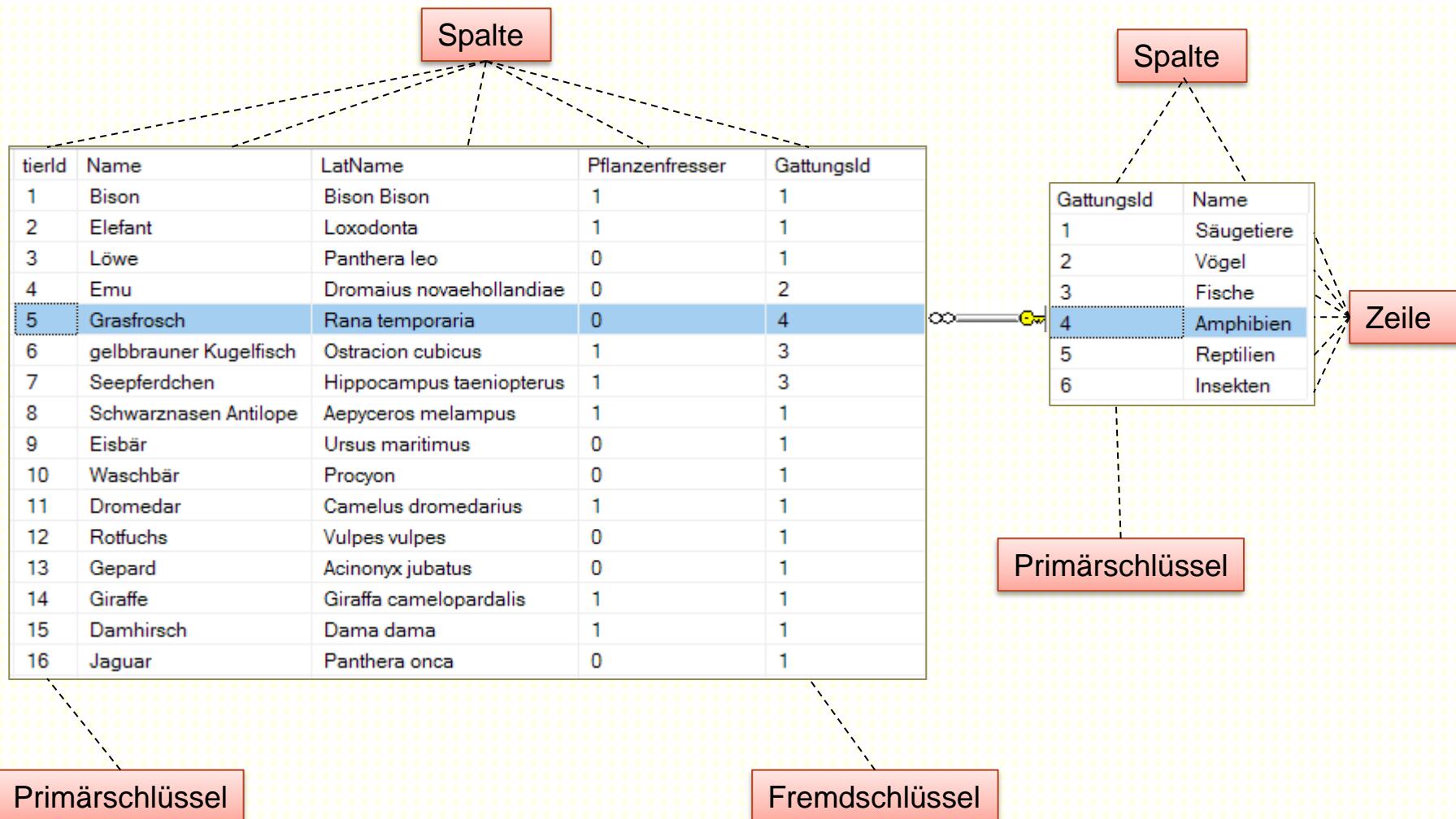
# **Datenbankzugriff mit JDBC**

# Relationale Datenbank

- **Tabelle (Relation)**
  - Einheit, in der Daten abgelegt werden
  - hat vordefiniertes Schema (Spaltendefinition)
- **Beziehungen (Assoziationen)**
  - definieren eine logische Verbindungen zwischen Tabellen



# Relationale Datenbank



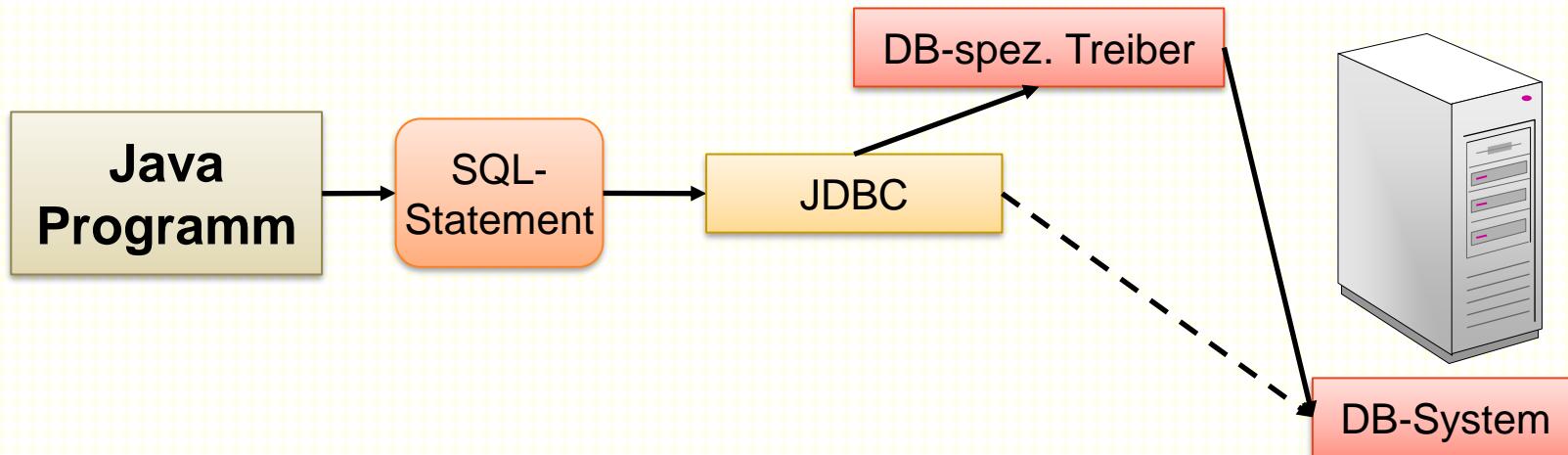
# *SQL – Structured Query Language*

- **Abfragesprache für Relationale Datenbanken**
  - nicht Case-Sensitive
  - definiert Befehle
    - zum Abfragen, Einfügen, Ändern und Löschen von Daten
      - Select, Insert, Update, Delete
    - zum Erstellen / Manipulieren von Tabellen Schemas
      - Create Table ...
  - ist standardisiert, der Standard wird von den meisten Herstellern nicht vollständig eingehalten

```
select tierId, Name, LatName, Pflanzenfresser, GattungsId  
from Tiere  
where Pflanzenfresser = 1  
order by Name
```

- **Java Database Connectivity**

- Standard Java API zur Anbindung von Java-Programmen an Datenbanken
- definiert die Schnittstellen für die erforderlichen Objekte
- Datenbank-spezifischer Treiber implementiert die Schnittstellen und stellt die eigentliche Anbindung bereit



# **Connection URLs**

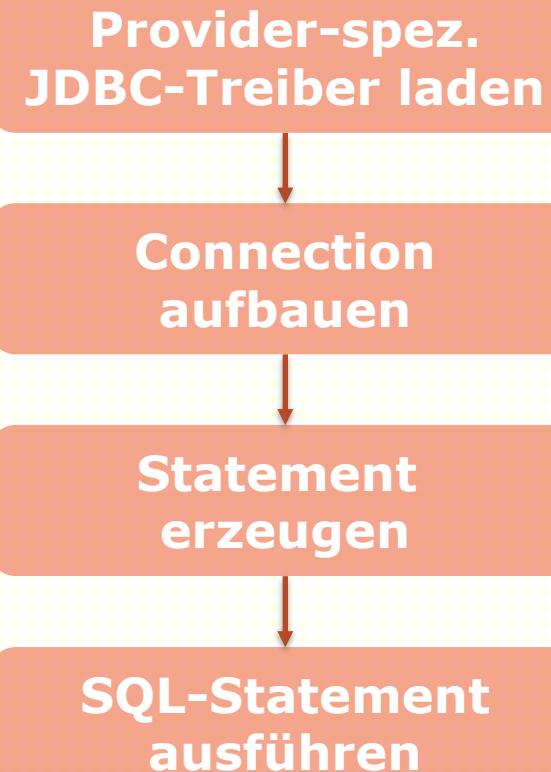
- **Connection (Verbindung)**
  - ermöglicht Zugriff auf eine Datenbank
  - zum Öffnen sind meist Anmeldeinformationen erforderlich
- **Connection URLs (Verbindungs-URL)**
  - enthalten alle erforderlichen Informationen um eine Verbindung zu einer Datenbank herzustellen
  - Syntax: `jdbc:<subprotocol>:<subname>`
    - `jdbc:` fix, gibt JDBC als Protokoll an
    - `<subprotocol>`: der Name des Providers, z.B. mysql
  - `<subname>`: Provider spezifische Informationen zur Verbindung
  - je nach Provider müssen ggf. Anmeldeinformationen separat angegeben werden

# **JDBC - Connection URLs**

| DB         | Sub protocol | Example                                                               |
|------------|--------------|-----------------------------------------------------------------------|
| Oracle     | oracle:thin  | jdbc:oracle:thin:@localhost:1521:orcl                                 |
| MySQL      | mysql        | jdbc:mysql://localhost/Zoo                                            |
| MariaDB    | mariadb      | jdbc:mariadb://localhost/Zoo                                          |
| DB2        | db2          | jdbc:db2:Zoo                                                          |
| MS SQL     | sqlserver    | jdbc:sqlserver://localhost:databaseName=Zoo; integratedSecurity=true; |
| Derby      | derby        | jdbc:derby://localhost:1527/Zoo                                       |
| PostgreSQL | postgresql   | jdbc:postgresql://localhost:5432/Zoo                                  |
| ODBC       | odbc         | jdbc:odbc:Zoo                                                         |

# JDBC – wichtige Klassen

- **DriverManager**
  - liefert zu einer Connection-URL ein passendes Connection-Objekt
  - (für sehr alte Treiber zuerst Class.forName(...) aufrufen)
- **Connection**
  - DB-Verbindung, gibt Zugriff auf Statement-Objekte
- **Statement**
  - Führt SQL-Befehle (Queries) an der Connection aus
- **ResultSet**
  - gibt Zugriff auf das Ergebnis eines Select-Befehls



# JDBC Beispiel

```
try (Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/zoo", "root", "") ) { {
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(
        "Select gattungsId, name from gattungen");
    while (rs.next()) {
        System.out.printf("%s - %d\n",
            rs.getString(2), rs.getInt(1));
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

Über das ResultSet können die Datensätze  
gelesen werden

# JDBC PreparedStatement

- **PreparedStatement**

- Vorkompiliertes Statement
- Query-Text wird bei Erzeugung angegeben
- Unterstützt Parameter

```
PreparedStatement pstmt = conn.prepareStatement(  
    "Select tierId, name, latName, imZooSeit from " +  
    " tiere where gattungsId = ?");  
pstmt.setInt(1, gattungsId);  
ResultSet rs = pstmt.executeQuery();  
while (rs.next()) {  
    System.out.printf("%s - %d - %s - %s \n",  
        rs.getString("name"), rs.getInt("tierId"),  
        rs.getString("latName"), rs.getDate("imZooSeit"));  
}
```

# JDBC Aktualisierung und Transaktionen

- **Statements unterstützen 2 Arten der Ausführung**
  - executeQuery: für Select-Statements, liefert ResultSet
  - executeUpdate: für Insert/Update/Delete Statements
- **Transaktion**
  - fasst mehrere Aktualisierungsbefehle zu einer logischen Einheit zusammen
  - die Befehle werden
    - entweder vollständig und fehlerfrei gespeichert (commit)
    - oder überhaupt nicht gespeichert, im Fehlerfall werden alle bisherigen Änderungen rückgängig gemacht (rollback)
  - sorgt für Konsistenz bei Aktualisierungen

# JDBC Aktualisierung und Transaktionen

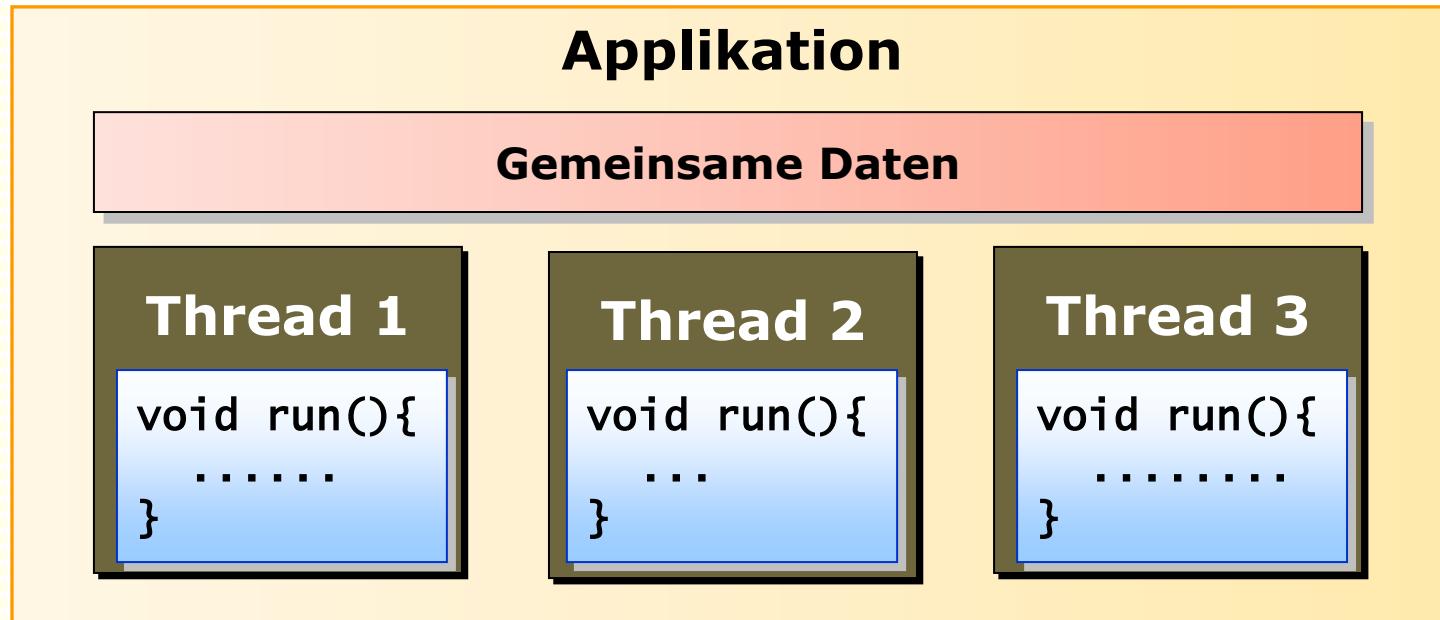
- **autoCommit-Eigenschaft der Connection**
  - steuert die Handhabung von Transaktionen
  - **autoCommit = true**
    - Automatische Transaktionssteuerung
    - Transaktion wird automatisch vor jedem Statement gestartet
      - im OK-Fall wird Commit ausgeführt
      - im Fehlerfall wird Rollback durchgeführt
    - eignet sich nur für einzelne Befehle
  - **autoCommit = false**
    - Manuelle Transaktionssteuerung
    - Transaktion wird von JDBC bei Bedarf automatisch gestartet
    - Programm führt commit oder rollback selber aus
    - wenn die Connection geschlossen wird, wird von vielen DB-Treibern eine offene Transaktion automatisch zurückgesetzt

# ***Nebenläufige Programmierung***

# Nebenläufige Programmierung

## ▪ Multithreading

- ermöglicht (quasi)parallele Ausführung
- mehrere parallel arbeitende Threads teilen sich Adressraum und Prozessorleistung
- Einige Betriebssysteme unterstützen Threads direkt



# Multithreading

- **Vorteile**

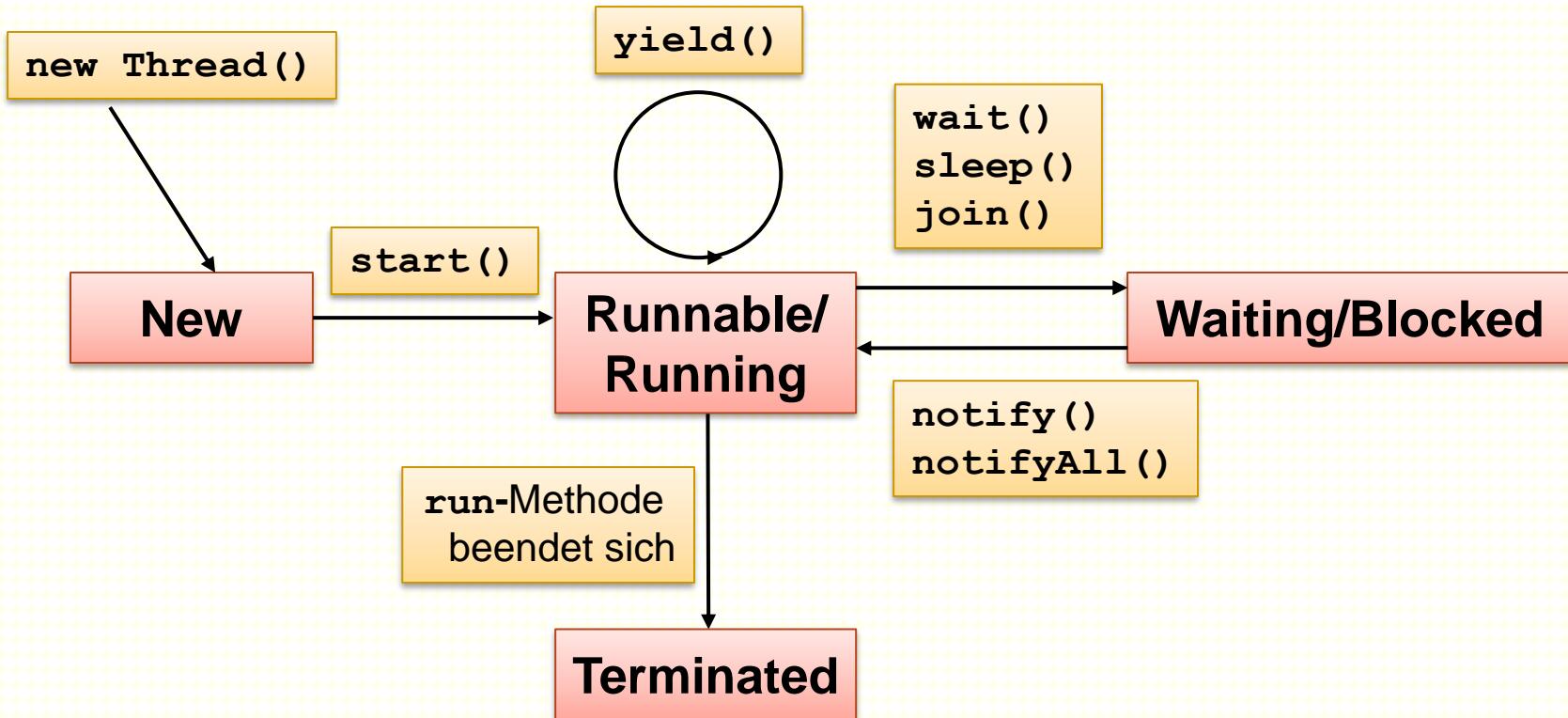
- UI kann während Hintergrundarbeiten auf User Input reagieren
- Verbesserte Performance durch parallele Ausführung
- Unterschiedliche Priorität möglich

- **Nachteile**

- vielfältige Fehlermöglichkeiten bei der Koordination und Synchronisation zwischen mehreren Threads

# Thread

## ▪ Zustände eines Threads



# Threads erzeugen I

- **Implementieren von Runnable**

```
class CounterThread implements Runnable {  
    @Override public void run() {  
        for (int i = 0; i < 5; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

- **Starten**

```
CounterThread runnable1 = new  
CounterThread();  
  
Thread t1 = new Thread(runnable1);  
t1.start();
```

Erzeugen mit einem  
Runnable-Objekt

Führt die run-Implementierung des Objekts aus

# Threads erzeugen II

- **Erweitern von Thread**

```
class CounterThread extends Thread {  
    @Override public void run() {  
        for (int i = 0; i < 5; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

- **Starten**

```
CounterThread t1 = new CounterThread();  
t1.start();
```

Erzeugen einer Instanz  
der abgeleiteten Klasse

Führt den run-Override des Objekts aus

# *Thread Eigenschaften*

- **getName/setName**
  - Name des Threads
  - Default: "Thread" + fortlaufende Nummer
- **getPriority/setPriority**
  - Priorität des Threads (zwischen Thread.MIN\_PRIORITY und Thread.MAX\_PRIORITY)
  - Default: die Priorität des erzeugenden Threads
- **getState**
  - Zustand des Threads (NEW, RUNNABLE, ...)
- **isDaemon/setDaemon**
  - ob der Thread ein Hintergrund-Thread ist
  - Hintergrundthreads halten die Applikation nicht am Laufen

# Thread Methoden

- **static void sleep(...)**  
**throws InterruptedException**
  - der aktuelle Thread wird für n Millisekunden blockiert
- **void join()**  
**throws InterruptedException**
  - auf das Ende eines Threads warten
  - overload mit Timeout
- **static Thread currentThread()**
  - liefert den aktuellen Thread, d.h. den Thread in dem der laufende Code ausgeführt wird

# Thread beenden

- **void interrupt()**
  - wenn der Thread blockiert ist oder wartet, wird eine **InterruptedException** ausgelöst
  - ist der Thread runnable, wird das **interrupted**-Flag gesetzt
- **boolean isInterrupted()**
  - liefert den Interrupted Status des Threadobjekts
- **static boolean interrupted()**
  - liefert den Interrupted Status des aktuellen Threads und setzt ihn zurück

# Thread beenden I

```
class EndlessRunnable implements Runnable {  
    @Override public void run() {  
        try {  
            while (! Thread.interrupted()) {  
                ...  
            }  
        } catch (InterruptedException e) {  
            // OK, dem Thread wurde das Ende angezeigt  
        }  
    }  
}
```

static Methode liefert interrupted Flag für den aktuellen Thread

```
Thread t1 = new Thread(new EndlessRunnable());  
t1.start();  
...  
t1.interrupt(); // den Thread beenden
```

# Thread beenden II

Instanzmethode liefert das interrupted Flag für ein Thread-Objekt

```
class EndlessThread extends Thread {  
    @Override public void run() {  
        try {  
            while (! isInterrupted()) {  
                ...  
            }  
        } catch (InterruptedException e) {  
            // OK, dem Thread wurde das Ende angezeigt  
        }  
    }  
}
```

```
EndlessThread t1 = new EndlessThread();  
t1.start();  
...  
t1.interrupt(); // den Thread beenden
```

# *Thread Synchronisation*

- **Probleme beim Zugriff auf gemeinsame Daten**
  - Wettlaufsituation (Race condition)
    - mehrere Threads verändern unkontrolliert gemeinsame Daten
  - Verklemmung (Deadlock)
    - Threads warten auf Objekte, die der jeweils andere blockiert
- **Beste Strategie**
  - Gemeinsamen Zugriff vermeiden
- **Möglichkeiten der Synchronisation in Java**
  - synchronized Methoden
  - synchronized Codeblöcke (kritische Abschnitte)
  - wait / notify

# **Synchronisation – synchronized method**

- **synchronized Instanzmethode**
  - synchronisiert Zugriff über eine Instanz
  - für **1 Instanz** darf gleichzeitig nur **1 Thread** eine der synchronized Methoden ausführen
  - alle anderen Threads sind blockiert, bis dieser fertig ist
- **synchronized static Methode**
  - synchronisiert den Zugriff über die Klasse
  - nur **1 Thread** darf gleichzeitig eine der static synchronized Methoden ausführen
  - alle anderen Threads sind blockiert, bis dieser fertig ist

# Synchronisation – synchronized method

Beispiel: Zugriff auf Instanz-Methoden synchronisieren

```
public class SafeCounter {  
    private int counter; // Zähler ist ein Instanzfeld  
    public synchronized void incrementCounter() {  
        counter++;  
    }  
    public synchronized void decrementCounter() {  
        counter--;  
    }  
    public synchronized int getCounter() {  
        return counter;  
    }  
}
```

Zugriff auf die Methode  
ist über das aktuelle  
Objekt geschützt

# Synchronisation – kritische Abschnitte

- **synchronized Statement**

- synchronisiert den Zugriff über ein Synchronisations-Objekt
- nur **1 Thread** darf Code, der **über dieses Objekt** geschützt ist, gleichzeitig ausführen
- alle anderen Threads sind blockiert, bis dieser fertig ist

Beispiel: Zugriff auf Instanz-Felder synchronisieren

```
public class SafeCounter {  
    public void increase () {  
        ...  
        synchronized (this) {  
            counter ++;  
        }  
    }  
    public void decrease () {  
        ...  
        synchronized (this) {  
            counter --; /  
        }  
    }  
}
```

Zugriff auf den Codeblock ist über das aktuelle Objekt geschützt

# Synchronisation – kritische Abschnitte

Beispiel: Zugriff auf statische Felder synchronisieren

```
public class SafeCounter {  
    private static final Object syncObject = new Object();  
    private static int staticCounter; // Zähler ist static  
    public static void increaseStatic () {  
        ...  
        synchronized(syncObject) {  
            staticCounter++;  
        }  
    }  
    public static void decreaseStatic () {  
        synchronized(syncObject) {  
            staticCounter--;  
        }  
    }  
}
```

Zugriff auf den Codeblock ist über ein unveränderliches static Objekt geschützt

# Synchronisation – kritische Abschnitte

Beispiel: Zugriff auf statische Felder synchronisieren

```
public class SafeCounter {  
    private static final Object syncObject = new Object();  
    private static int callCounter; // Zähler ist static  
    public void doSomething () {  
        ...  
        synchronized(syncObject) {  
            callCounter ++;  
        }  
    }  
    public void doSomethingElse () {  
        synchronized(syncObject) {  
            callCounter --;  
        }  
    }  
}
```

Auch in Instanzmethoden kann ein statisches Objekt für die Synchronisation erforderlich sein

# *wait and notify*

- **Synchronisierte Abfolge von Aktionen**
  - mit Methoden aus der Klasse Object
  - **obj.wait()**
    - hält einen laufenden Thread an
    - Sperre für obj wird freigegeben
  - **obj.notify() , obj.notifyAll()**
    - löst einen / alle wartenden Threads aus
  - muss in kritischem Abschnitt / in synchronized Methode aufgerufen werden, der/die über obj geschützt ist
    - bei Verstoß wird eine IllegalMonitorStateException ausgelöst

# *wait and notify*

Beispiel: Synchronisation zwischen Sender und Empfänger

```
public class Transfer {  
    private String buffer;  
    public synchronized void liefern(String msg) throws ... {  
        while (buffer != null) // warten bis Platz  
            wait();  
        this.buffer = msg;  
        notifyAll(); // andere Threads informieren  
    }  
    public synchronized String abholen() throws ... {  
        while (buffer == null) // warten bis Nachricht da ist  
            wait();  
        String message = this.buffer; // Nachricht auslesen  
        this.buffer = null; // den Puffer zurücksetzen  
        notifyAll(); // andere Threads informieren  
        return message; // Nachricht zurückliefern  
    }  
}
```

# *wait and notify*

Beispiel: Synchronisation zwischen Sender und Empfänger

```
final Transfer transfer = new Transfer();
Runnable sender = ()-> { // Runnable-Implementierung für Sender
    for (int i = 1; true; i++)
        try {
            transfer.liefern("Nachricht " + i); // Nachricht senden
        } catch (InterruptedException e) { break; }
};
Runnable recvr = ()-> { // Runnable-Implementierung für Empfänger
    while (true)
        try {
            String msg = transfer.abholen(); // Nachricht empfangen
        } catch (InterruptedException e) { break; }
};
Thread[] threads = { new Thread(sender), new Thread(recvr) };
for (Thread thread : threads)
    thread.start(); // Sender und Empfänger parallel ausführen
```

# *Executor und ExecutorService*

- **Neue Interfaces zur Abstraktion der parallelen Ausführung**
  - Thread für die Ausführung wird aus einem Thread-Pool bereitgestellt
  - Systemressourcen können effizienter genutzt werden
  - **Executor**
    - für die Ausführung von Runnable-Objekten
  - **ExecutorService**
    - ermöglicht zusätzlich die Ausführung von Callable-Objekten
  - **Callable<T>**
    - definiert eine Methode die einen Wert berechnet: `T call()`
  - **Future<T>**
    - repräsentiert ein zukünftiges Ergebnis

# *ExecutorService Instanz*

- **ForkJoinPool**
  - `commonPool()`: gibt Zugriff auf den Standard-Thread-Pool
- **Hilfsklasse Executors stellt Thread Pools bereit**
  - Executor muss nach der Verwendung beendet werden

| Methode                                                    | Liefert                                                |
|------------------------------------------------------------|--------------------------------------------------------|
| <code>newSingleThreadExecutor()</code>                     | Executor, der immer nur einen Task ausführt            |
| <code>newFixedThreadPool<br/>(int nThreads)</code>         | Executor mit fixer Anzahl von Threads                  |
| <code>newCachedThreadPool()</code>                         | Executor, der neue Threads nach Bedarf erzeugt         |
| <code>newScheduledThreadPool<br/>(int corePoolSize)</code> | Executor, bei dem Tasks zeitlich geplant werden können |

# ExecutorService Beispiel

```
Callable<Integer> task = () -> {
    int result = 0;
    for (int i = 1; i <= 1000; i++) {
        result += ...; // aufwändige Berechnung
    }
    return result;
};

ExecutorService exec = Executors.newSingleThreadExecutor();
Future<Integer> future1 = exec.submit(task); // parallel
Future<Integer> future2 = exec.submit(() -> { ... }); // parallel
try {
    System.out.println("Tasks gestartet, warte auf Ergebnis");
    Integer result1 = future1.get(), result2 = future2.get();
    ...
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
exec.shutdown(); // Threadpool beenden
```

# Asynchrone Tasks

- **CompletableFuture<T>**
  - kapselt einen Task, der asynchron ausgeführt wird
  - ermöglicht einfaches Starten und Verketten von Tasks
  - statische Methoden erzeugen und starten Tasks
    - `runAsync`: Task mit `Runnable`-Implementierung ausführen
    - `supplyAsync`: Task mit `Supplier<T>`-Implementierung ausführen
  - Instanzmethoden verketten und verarbeiten Tasks, z.B.:
    - `thenAccept`: das Ergebnis verarbeiten
    - `thenRun`: nach dem Ende des Tasks eine Aktion ausführen
    - `get`: auf das Ergebnis warten
    - `cancel`: den Task mit `CancellationException` abbrechen
    - `isDone`, `isCancelled`: prüft den Status des Tasks

# Asynchrone Tasks Beispiel

```
Supplier<Integer> task = () -> {
    int result = 0;
    for (int i = 1; i <= 1000; i++) {
        result += ...; // aufwändige Berechnung
    }
    return result;
};

CompletableFuture<Integer> async1, async2;
// Tasks erzeugen und asynchron starten
async1 = CompletableFuture.supplyAsync(task); // parallel
async2 = CompletableFuture.supplyAsync(() -> { ... }); // parallel
// mit BiConsumer die beiden Ergebnisse verarbeiten
async1.thenAcceptBoth(async2,
    (r1, r2) -> System.out.printf("erg1=%d, erg2=%d\n", r1, r2));
```

# ***Netzwerk- Programmierung***

- **Server-Client-Kommunikation mit Sockets**
  - direkte TCP/IP Verbindung zwischen 2 Rechnern
  - Klasse ServerSocket: für serverseitiges Warten auf Client-Verbindung ("accept")
  - Klasse Socket: für die Netzwerkkommunikation zwischen einem Client und einem Server
    - Clientseitig fürs Herstellen der Verbindung ("connect")
    - Serverseitig für die Kommunikation mit einem Client
  - Datenaustausch erfolgt über 2 Byte-Streams
    - InputStream Socket.getInputStream zum Lesen
    - OutputStream Socket.getOutputStream zum Schreiben
    - können mit anderen Streams verkettet werden

# Sockets

```
1 ServerSocket srv =  
    new ServerSocket(4321);  
  
2 // auf Client warten  
Socket client = srv.accept();  
  
3 // Streams holen  
InputStream inStream =  
    client.getInputStream();  
OutputStream outStream =  
    client.getOutputStream();  
  
4 inStream.read(...);  
outStream.write(...);  
  
5 // Socket schließen  
client.close();  
  
6 ...  
srv.close();
```

```
// Verbindung öffnen  
Socket conn = new Socket  
    ("localhost", 4321);  
  
// Streams holen  
InputStream inStream =  
    conn.getInputStream();  
OutputStream outStream =  
    conn.getOutputStream();  
  
outStream.write(...);  
inStream.read(...);  
  
...  
// Socket schließen  
conn.close()
```

# *HTTP Requests*

- **URL – Uniform Resource Locator**
  - dient zur Adressierung  
`http://server.mycompany.at:8080/zoo/index.html`
- **HTTP Request**
  - kann Header enthalten, z.B. für
    - User-Languages, User-Agent etc.
  - wird mit einer Method (auch Verb genannt) gesendet, z.B.:
    - GET, PUT, POST, DELETE, HEAD, etc
    - bei einigen Methods können zusätzlich zum URL weitere Daten gesendet werden

# *HTTP Requests*

- **HTTP Response**
  - enthält einen Status Code:
    - 2xx: OK (z.B. 200 OK)
    - 4xx: Client-Fehler (z.B. 404 Nicht gefunden)
    - 5xx: Server-Fehler (z.B. 500 Internal Server Error)
  - kann beliebigen Content enthalten, z.B.
    - HTML, Grafiken, CSS, JavaScript, PDF etc.
  - enthält Header, z.B. für
    - Content-Type, Content-Length

# *HTTP Requests mit URLConnection*

- **Klasse URL**
  - Kapselt einen (HTTP) URL
  - openStream(): liefert den InputStream für den Download
  - openConnection(): liefert URLConnection
    - Request kann angepasst werden (z.B. Header)
- **Klasse URLConnection**
  - Handelt Up- und Download von HTTP Requests
    - getInputStream() liefert den InputStream für den Download
    - getOutputStream() liefert den OutputStream für den Upload von Daten (PUT, POST)
  - Informationen über den Response:
    - getHeaderFields(): gibt Zugriff auf die Header-Felder
    - getContentType(): liefert den MIME-Type
    - getContentLength(): liefert die Anzahl der Byte im Response

# *HTTP Requests mit URLConnection*

```
// passendes URL-Objekt erzeugen
URL url = new URL("http://www.mit.at");
// URLConnection öffnen
URLConnection conn = url.openConnection();
// Länge des Response
int length = conn.getContentLength(), read = 0;
// Stream zum Download holen
InputStream is = conn.getInputStream();
// Daten lesen
while (read < length) {
    ...
}
```

# *HTTP Requests mit der HttpClient API*

- **Neue API für HTTP Requests (seit Java 11)**
  - HttpRequest:
    - kapselt einen Request
    - wird durch Builder-Calls konfiguriert und erstellt
  - HttpResponse
    - kapselt die Antwort
  - BodyHandler
    - verarbeitet den eigentlichen Inhalt (den Body) des Response:
      - als gesamter String oder zeilenweise
      - als Byte-Array oder InputStream
      - als Download in File
  - HttpClient
    - sendet den Request unter Verwendung eines BodyHandlers

# *HTTP Requests mit der HttpClient API*

```
URI uri = new URI(url);
// Request erzeugen
HttpRequest req = HttpRequest.newBuilder(uri).GET()
    .setHeader("User-Agent", "Java Demoprogramm").build();
// Handler für den Body erzeugen
BodyHandler<String> handler =
    HttpResponse.BodyHandlers.ofString();
// Client erzeugen
HttpClient client = HttpClient.newHttpClient();
// Request senden
HttpResponse<String> response = client.send(req, handler);
// auf das Ergebnis zugreifen
String content = response.body();
System.out.println(content);
```