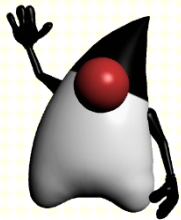


Java Grundlagen

DI Jürgen Wurzer, MSc



OpenJDK



tecTrain®



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 / Jakarta EE:**
 - Java/Jakarta 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

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) (LTS)	Neue Sprachfeatures (Lambda Expressions und Stream API); erste LTS Release, Support bis 2030
2017	Java 9	Open Source Referenzimplementierung

Jahr	Version	Anmerkung
2017	Java 9	Open Source Referenzimplementierung
2018	Java 11 (LTS)	LTS Release, Support bis 2024, Extended Support bis 2032
2021	Java 17 (LTS)	LTS Release, Support bis 2027, Extended Support bis 2029
2023	Java 21 (LTS)	aktuelle LTS Release, Support bis 2029, Extended Support bis 2031

▪ Seit Java 11

– Neuer Release-Zyklus

- Etwa alle 6 Monate
 - **STS Release (Short-Term-Support)**, Support endet mit nächster Release
- Etwa alle 3 Jahre bzw. derzeit alle 2 Jahre
 - **LTS Release (Long-Term-Support)**, 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/21/docs/api/index.html>

- Eclipse **Adoptium** (früher AdoptOpenJDK)

- <https://adoptium.net/>
 - stellt Installationspakete und regelmäßige Updates für Open JDK bereit / Binaries: „Eclipse Temurin“

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



Apache NetBeans

▪ IntelliJ Idea

- Relativ neue IDE der Firma JetBrains
 - Community Edition – Open Source, kostenlos
 - Ultimate Edition - kostenpflichtig



IntelliJ IDEA



Installation

▪ Mit Admin-Rechten

- Installationspakete 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

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 11**
 - gibt es nur noch ein JDK Paket
 - Adoptium bietet weiter JDK und JRE an

- **Java ist objektorientiert konzipiert**
 - Fordert Grundlegendes Verständnis für Klassen
- **Was sind ...**
 - ... Klassen?
 - ... Methoden?
 - ... Objekte?

Klassen

▪ Klasse

- ist eine Vorlage / Bauplan anhand der Objekte erzeugt werden können.
- ist ein Bauplan zur Erstellung einer Menge von Objekten
 - mit gleicher Struktur (Attributen) und
 - gleichem Verhalten (Methoden)

▪ In einer Klasse werden

- Attribute (=Daten, Zustand, Status) und
- Methoden (=Funktionalität, Verhalten) definiert

▪ Klassename

- in PascalCase / UpperCamelCase

▪ Dateiname

- muss gleich wie die Klasse inkl. .java lauten. E.g. HelloWorld.java

```
public class HelloWorld {  
    // Hier koennen Attribute und Methoden deklariert werden  
    // ...  
}
```

▪ Methode

- Ist ein Unterprogramm
- Heißt in Java Methode und nicht Funktion
- Muss innerhalb einer Klasse deklariert werden
- Methoden können statisch oder nicht statisch sein
 - Statisch:
 - Kann ohne Objekt der Klasse aufgerufen werden
 - Kein zugehöriges Objekt
 - Achtung: `static` in Java != `static` in C
 - Nicht statisch:
 - Methode ist an ein Objekt gebunden (per `this`-Referenz)
 - Aufruf über das Objekt
- Die `main`-Methode ist eine statische Methode
- Methodenname in lowerCamelCase

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!");  
    }  
}
```



Compiler

Bytecode

....
....
main
... ↓
.. ↓

Übersetzung erfolgt in einen einheitlich genormten Byte-Code

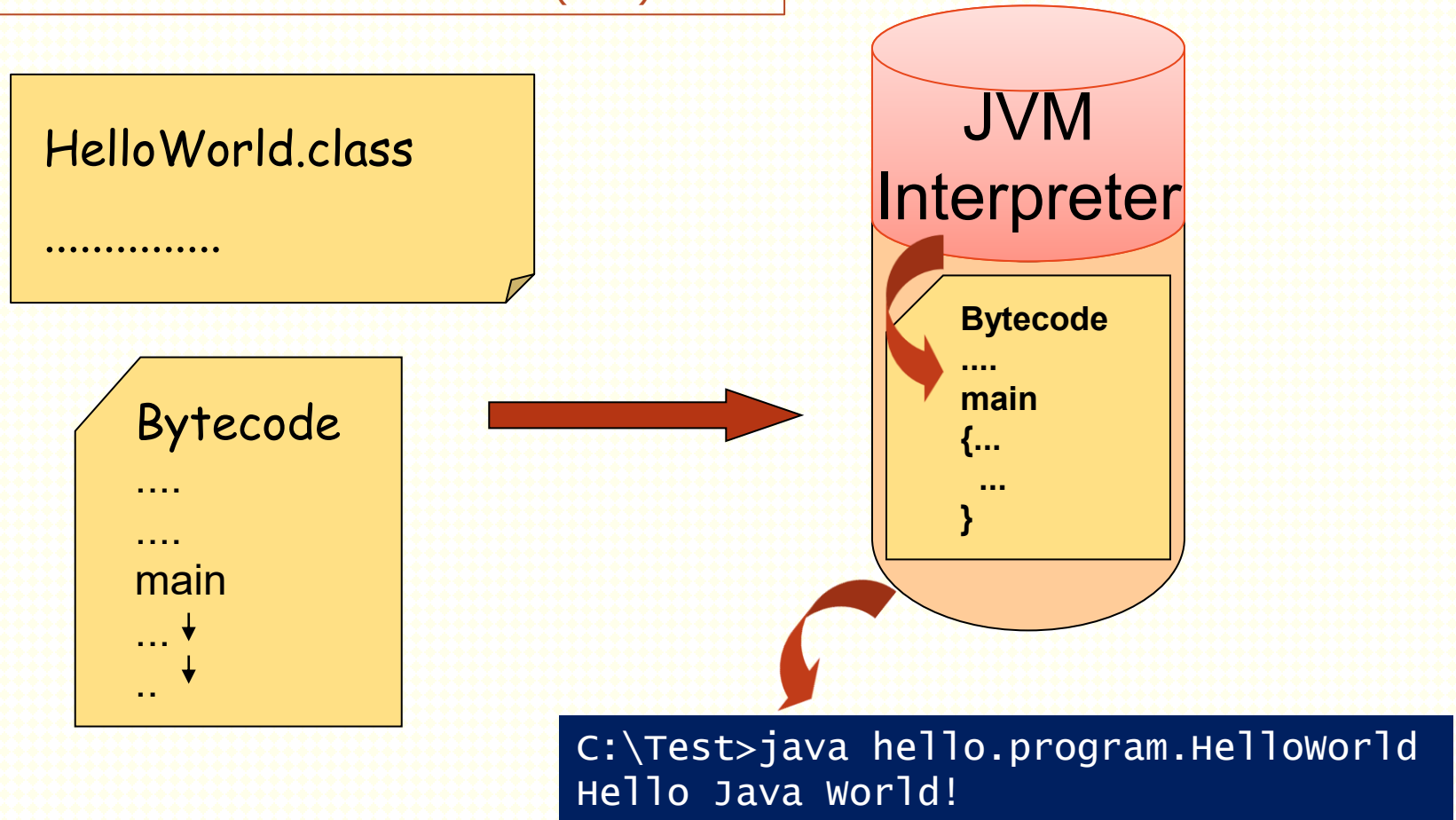
HelloWorld.class

.....

hello/program/HelloWorld.class

Ausführung

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



Kompilierung & Ausführung in der Konsole

▪ Java Compiler und Java Runtime Environment

```
javac
```

```
-d <directory>
```

```
-sourcepath <dir>
```

```
HelloWorld.java
```

Java Compiler

Verzeichnis für .class-Files

Verzeichnis für weitere Sourcefiles

Datei die kompiliert wird

```
java
```

```
-cp <dir>
```

```
hello.program.HelloWorld
```

Java Virtual Machine

Class Path: Klassen werden hier gesucht

Klasse deren main() ausgeführt wird

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;`

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

```
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);
```

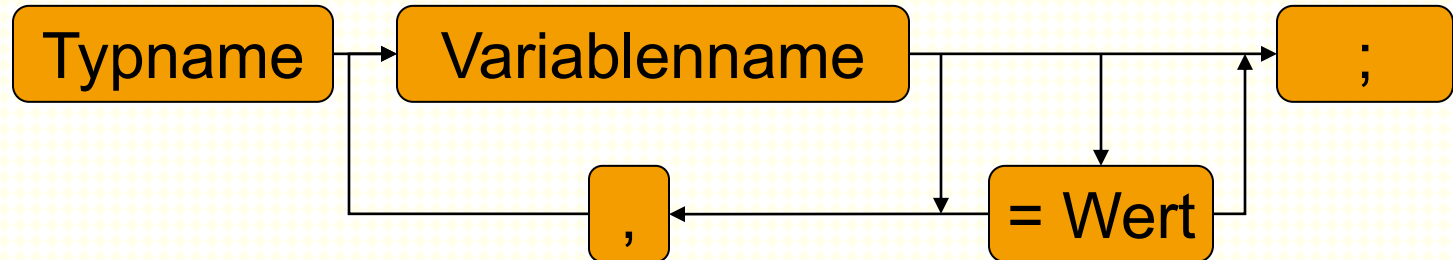
```
Name: "Elefant"
```

```
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
```


Operatoren

▪ Arithmetische

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo

▪ Vergleich

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

▪ In-, Dekrement

++	Inkrement
--	Dekrement

▪ Bitweise

&	UND
	inklusive ODER
^	exklusives ODER
~	Komplement

▪ Logische

&&	logisches UND
	logisches ODER
!	logisches NOT

Operatoren

▪ Shift

<<	nach links
>>	nach rechts (signed)
>>>	nach rechts (unsigned)

▪ Diverse

.	Memberzugriff
[]	Indezzugriff
? :	Bedingte Bewertung
(Typ)	type cast

▪ Zuweisung

=	Zuweisung
+= -= *= /= %=	Abkürzung für arithmetische Operatoren
<<= >>= >>>=	Abkürzung für Shiftoperatoren
&= = ^=	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

Beschreibung	Operator	Assoziativität
Access, Parentheses	[] . ()	→ L-to-R
Postfix	a++ a--	Not Associative
Unary	++a --a +a -a ~ !	← R-to-L
Cast, Creation	(type) new	← R-to-L
Multiplicative	* / %	→ L-to-R
Additive	+ -	→ L-to-R
Shift	<< >> >>>	→ L-to-R
Relational	< > <= >= instanceof	Not Associative
Equality	== !=	→ L-to-R
Bitwise AND	&	→ L-to-R
Bitwise XOR	^	→ L-to-R
Bitwise OR	 	→ L-to-R
Logical AND	&&	→ L-to-R
Logical OR	 	→ L-to-R
Ternary	? :	← R-to-L
Assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=	← R-to-L

niedrigste

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
- Formatierter String mit String.format() möglich
- Platzhaltersyntax z.B: `%1$+020.10f`

`%[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("%2$s %1$X\n", i, s);  
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  
Hey! 5A  
0090  
5,678000  
5,68  
05,68
```


■ Konsoleneingabe

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

- `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

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;  
}
```

Standardsyntax in allen
Java-Versionen

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

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 `zahl` gilt nur im Block der for-each-Schleife

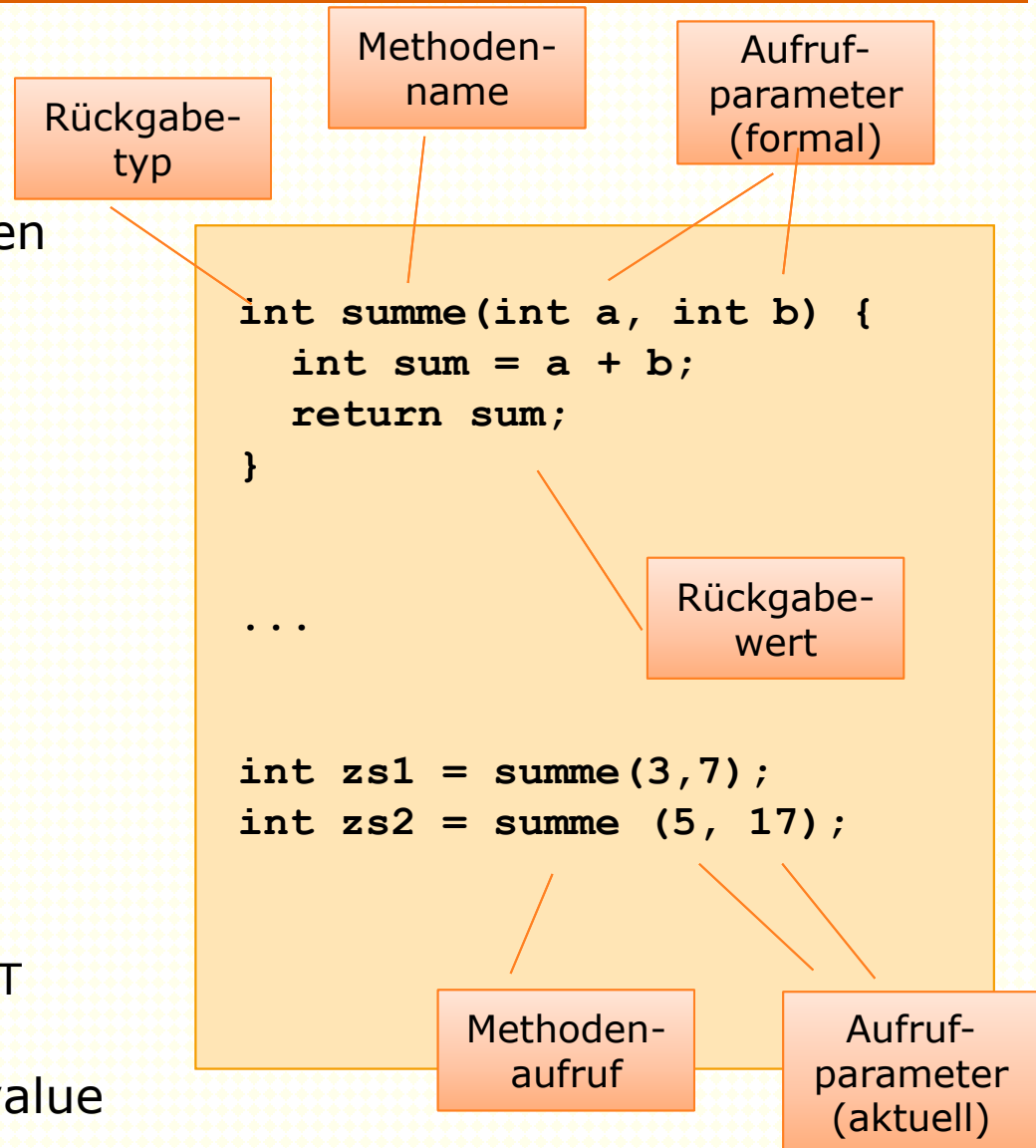
Kontrollstrukturen - Sprünge

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

Methode

▪ Definiert einen Unter-Algorithmus

- kann mehrfach aufgerufen werden
- kann beim Aufruf Argumente (Parameter) erhalten
- kann einen Wert zurückliefern
- Die formale Schnittstelle (Rückgabetypp, Name, Parametertypen) heißt **(Methoden-)Signatur**
 - Die Methoden-Signatur beinhaltet in Java NICHT den Rückgabetypp!
- Call by value / pass by value



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

▪ Methoden die sich selbst aufrufen

- Dürfen sich nicht unendlich oft selbst aufrufen (Stackoverflow)
- Benötigt eine Abbruchbedingung in dieser Funktion
- Auch der gegenseitige Aufruf von z.B. zwei Methoden stellt eine Rekursion dar.

```
public static void printNumbers(int nr) {  
    System.out.print(" " + nr);  
    if (nr > 0) {  
        printNumbers(nr - 1);  
    }  
}
```

← Rekursiver Aufruf

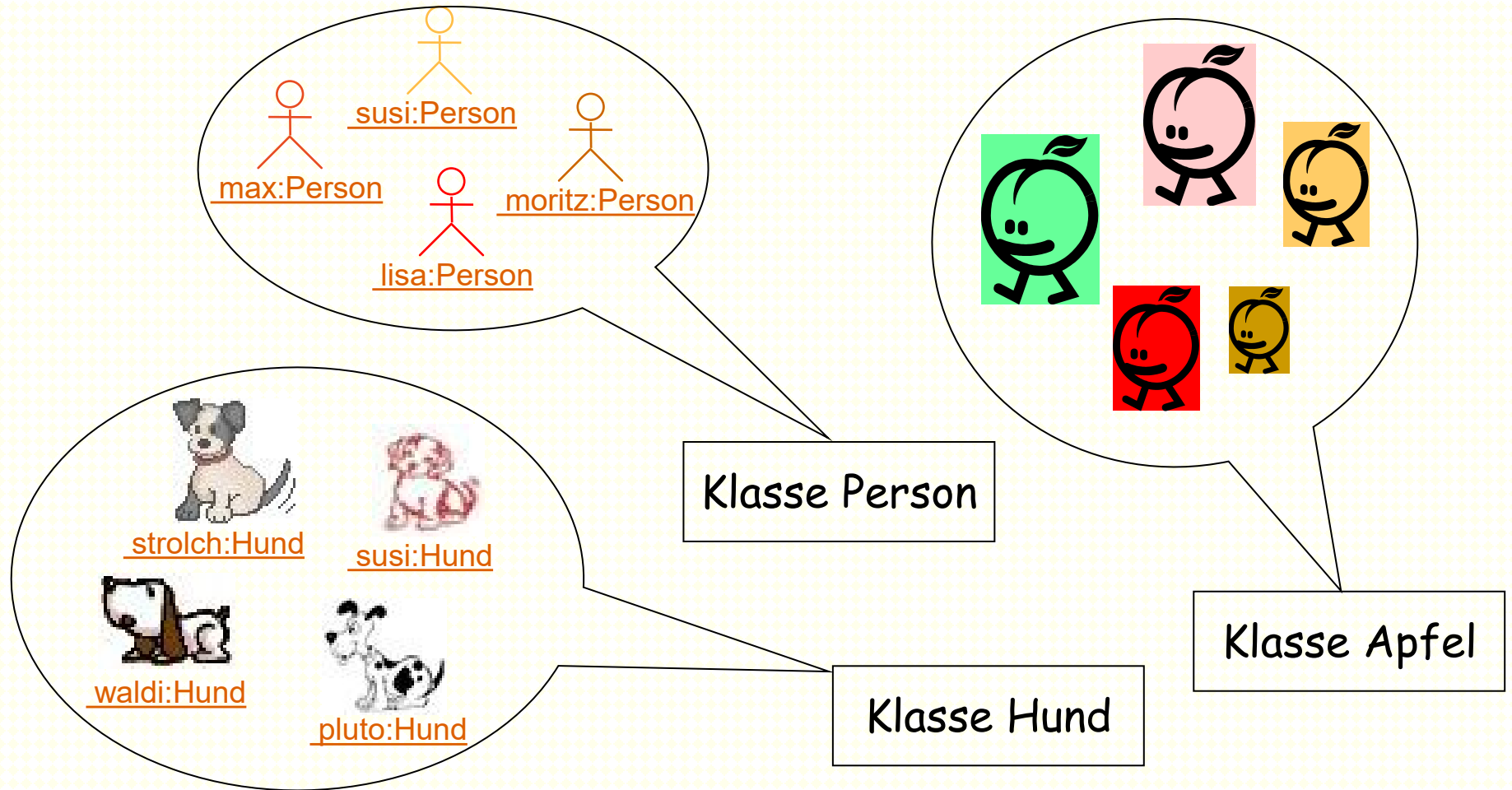
Aufruf z.B in main(): printNumbers(10);

Ausgabe: 10 9 8 7 6 5 4 3 2 1 0

Objektorientierte Konzepte

Grundlagen

Klassen und Objekte



Klassen und Objekte

**Klasse
(Class)**



Typ

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

**Objekt
(Object)**



Wert

- Exemplar ("Instanz") einer Klasse
- Beinhaltet konkrete Werte für die Attribute

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 ...;  
    }  
}
```

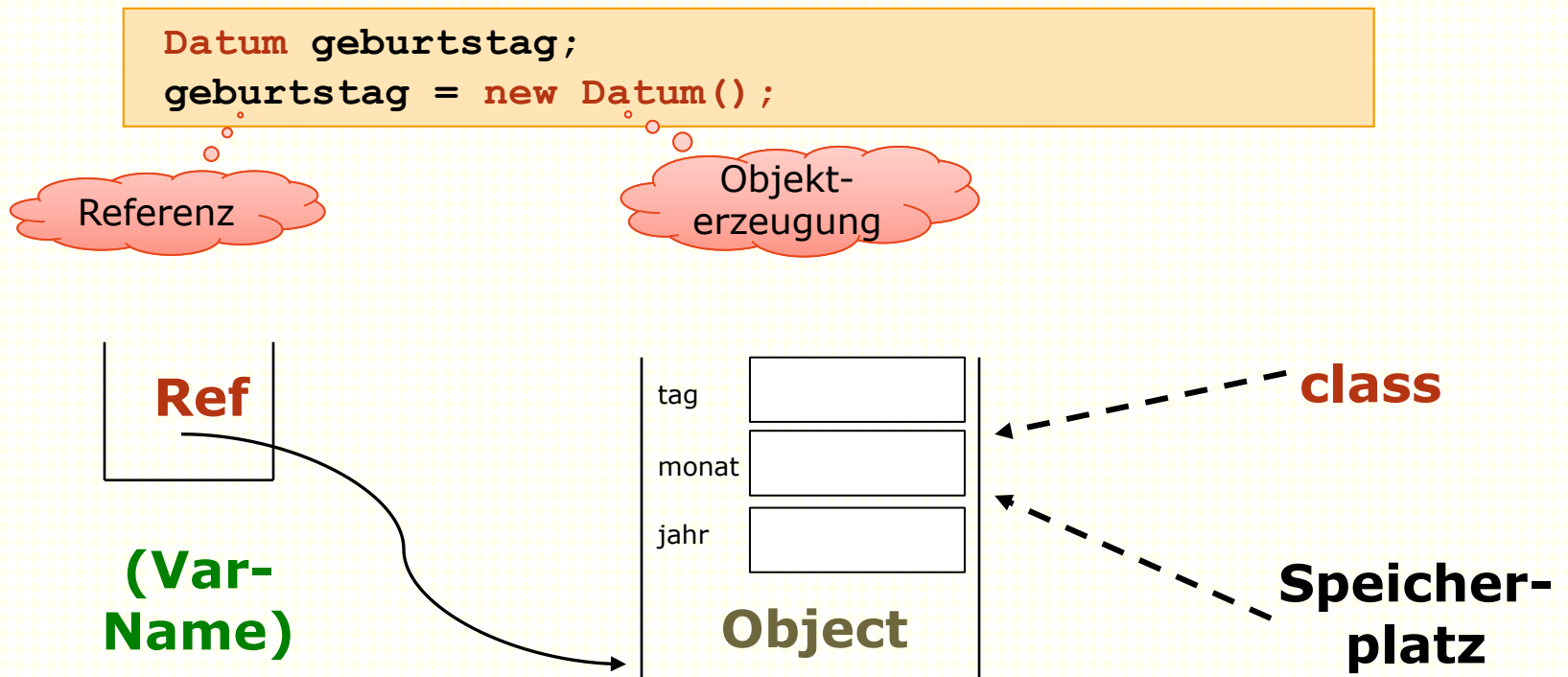
Attribute (oft "private"),
legen fest welche Daten
die Objekte haben

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

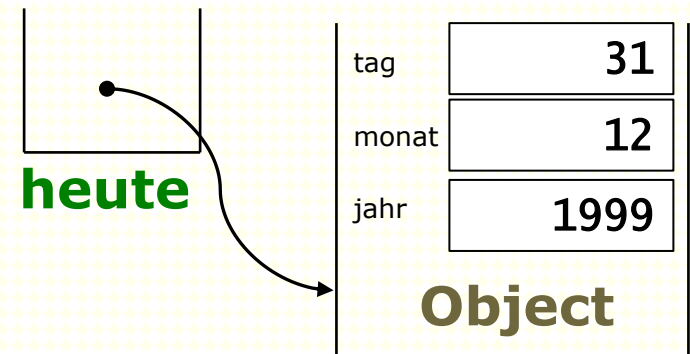
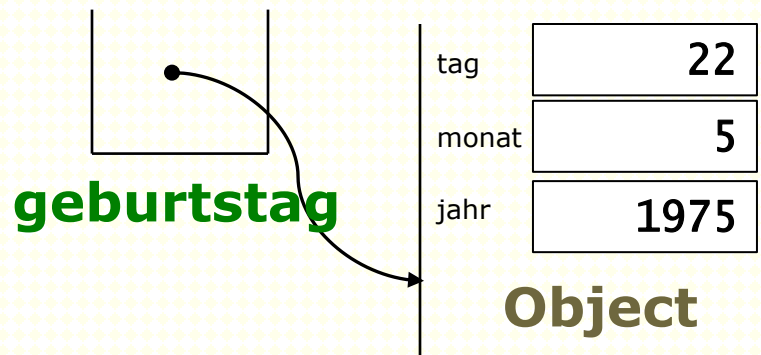
- erfolgt über eine Referenz
- mit dem Operator für den Memberzugriff .

```
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

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



*Welches Datum
enthält das
neue Objekt?*

- **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 Rückgabetyt

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 automatisch erzeugt**

- wenn die Klasse irgend einen Konstruktor enthält

Werttypen und Referenztypen

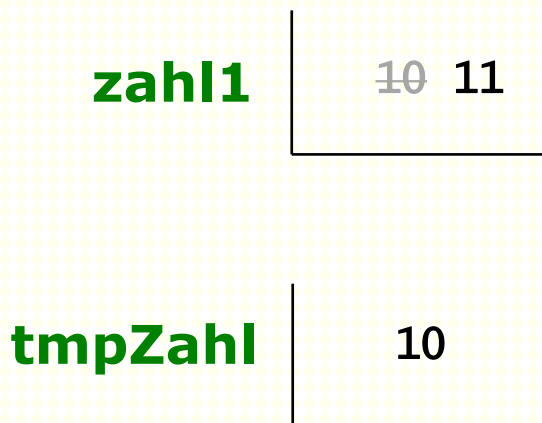
	Werttypen	Referenztypen
Allokation	am Stack	am Heap
Instanziierung	durch Deklaration	mit new
Zerstörung	am Ende des Blocks	Garbage Collector
Zuweisung	Kopie des Wertes	Kopie der Referenz
Parameter-Übergabe	Kopie des Wertes	Kopie der Referenz
Return-Wert	Kopie des Wertes	Kopie der Referenz
Vergleichsoperator == !=	Vergleichen den Wert	Vergleichen die Referenz *)
Betrifft	primitive Typen	„Referenztyp“, Klassen, Enums, Interfaces, Arrays

*) Wertvergleich für Referenztypen mit equals

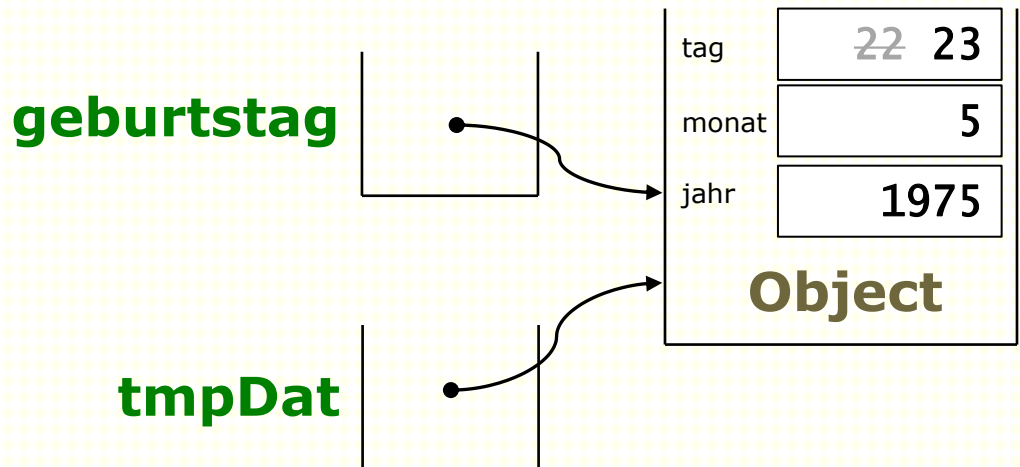
Werttypen und Referenztypen

- **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 ++;
```



```
Datum geburtstag, tempDat;  
geburtstag = new Datum(22, 5, 1975);  
tempDat = geburtstag;  
geburtstag.tagDazu(1);
```



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

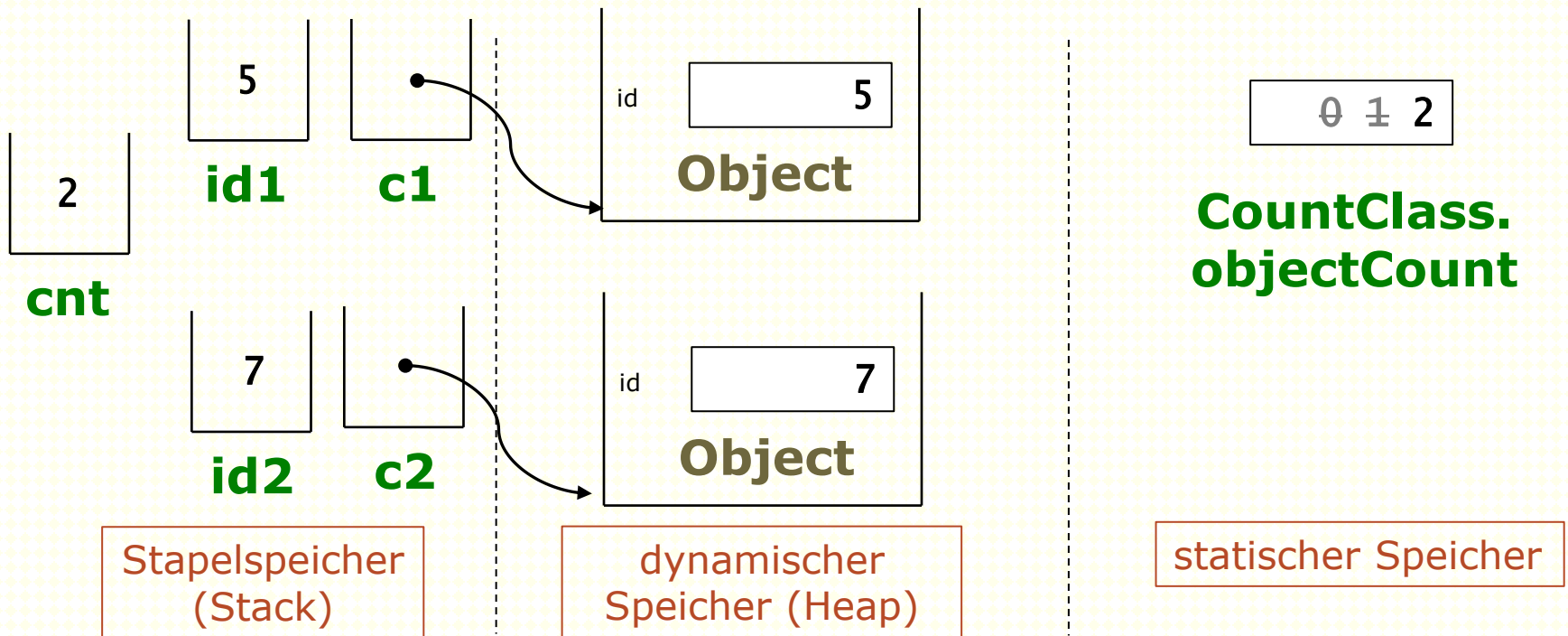
In static Methoden kann nur auf static Elemente zugegriffen werden

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 cnt = CountClass.getObjectCount();  
}
```

Der Zugriff auf Instanzmember erfolgt über eine Referenz

Der Zugriff auf statische Member erfolgt über den Klassennamen



Instanz- vs. Statische Member

	Instanzmember	Statische Member
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 static
Einsatz	<ul style="list-style-type: none">○ mehrere Exemplare der Klasse mit jeweils eigenen Werten;○ Nutzen fortgeschrittener OOP-Techniken	<ul style="list-style-type: none">○ Utility-Methoden, die ohne vorherige Instanziierung verwendbar sein sollen;○ "Globale" Methoden

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)

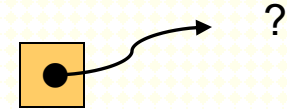
Array – Vektor

- **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

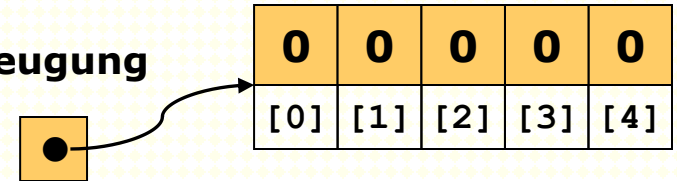
```
int[] zahlen;
```

Deklaration



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

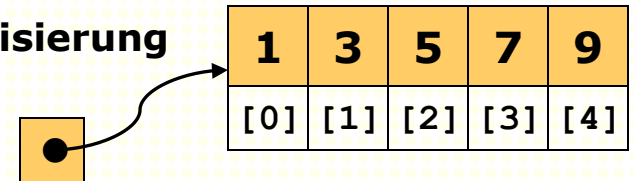
Erzeugung



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

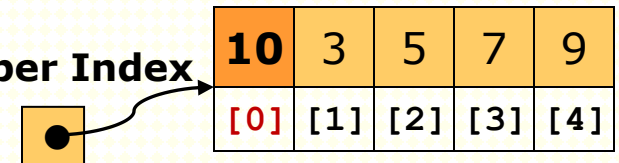
```
int[] zahlen = { 1, 3, 5, 7, 9 };
```

Initialisierung



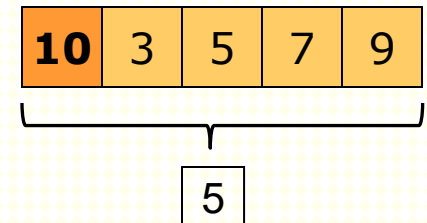
```
zahlen[0] = 10;
```

Zugriff per Index



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

Anzahl der Elemente



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 (seit Java 15)
- 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

▪ 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

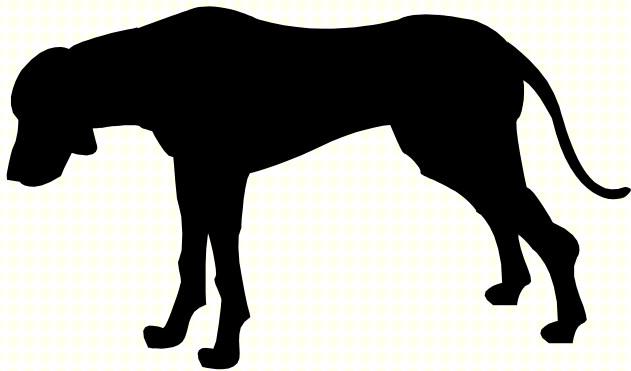
Methode	Zweck
length, charAt, indexOf, lastIndexOf	analog String
append, insert	Zeichenfolge oder primitiven Wert am Ende oder ab Index einfügen
delete, deleteCharAt	Zeichen von-bis bzw. am Index löschen
setLength	neue Länge setzen
replace	Zeichen von-bis durch andere Zeichenfolge ersetzen
reverse	die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

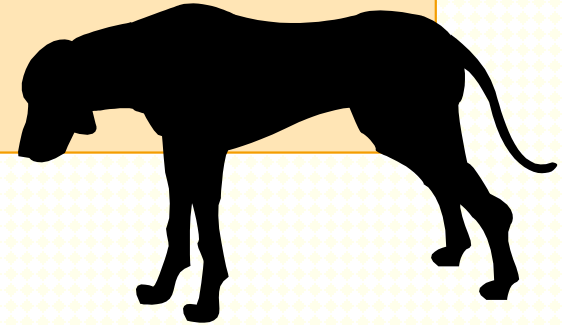
class Hund

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

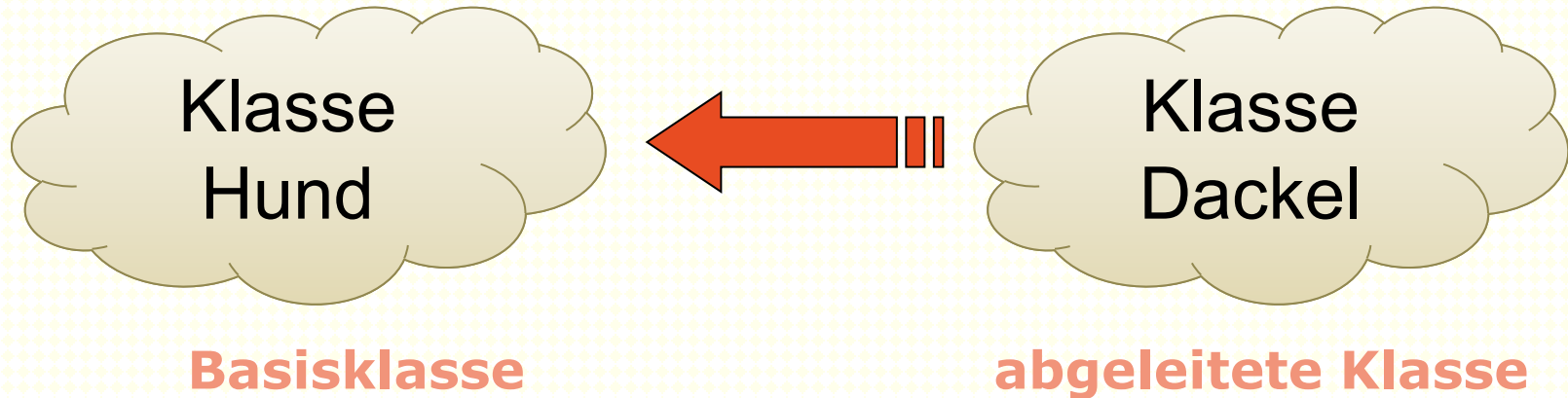
- **Falls der Basisklassen-Konstruktor Parameter hat**
 - werden die Parameter über einen **expliziten super-Aufruf** übergeben
 - Der super-Aufruf muss als **1. Anweisung** in einem Konstruktor stehen

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

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

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

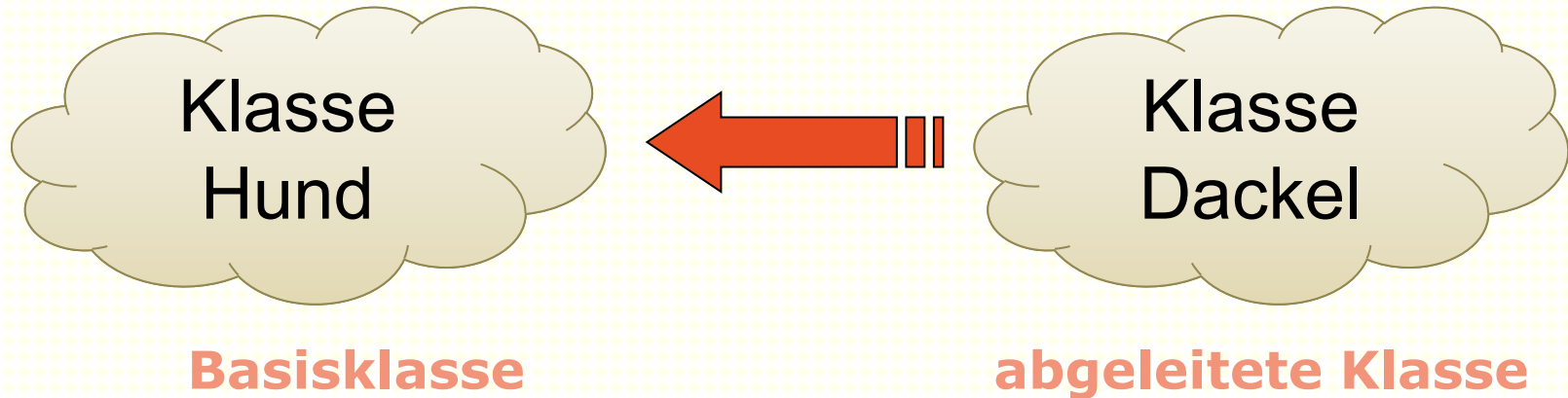
Vererbung – Typkompatibilität



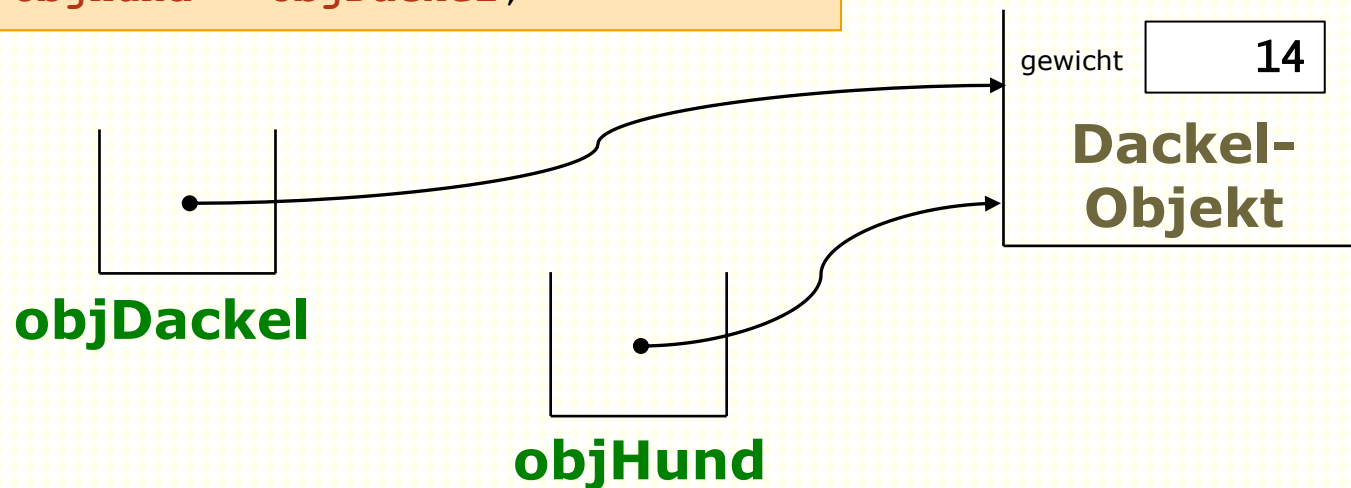
```
Dackel objDackel = new Dackel(14);  
Hund objHund = objDackel;
```

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

Vererbung – Typkompatibilität

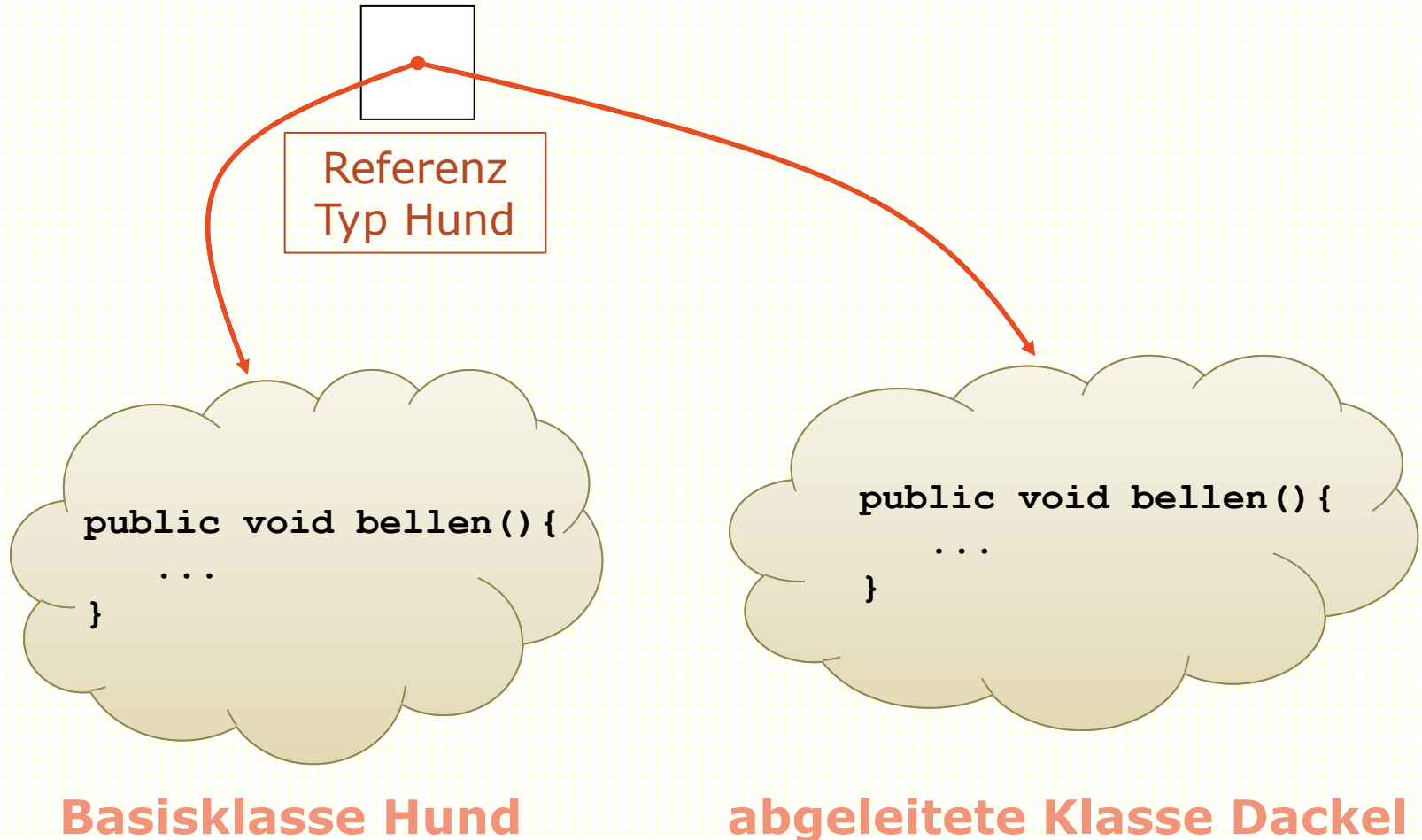


```
Dackel objDackel = new Dackel(14);  
Hund objHund = objDackel;
```



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**

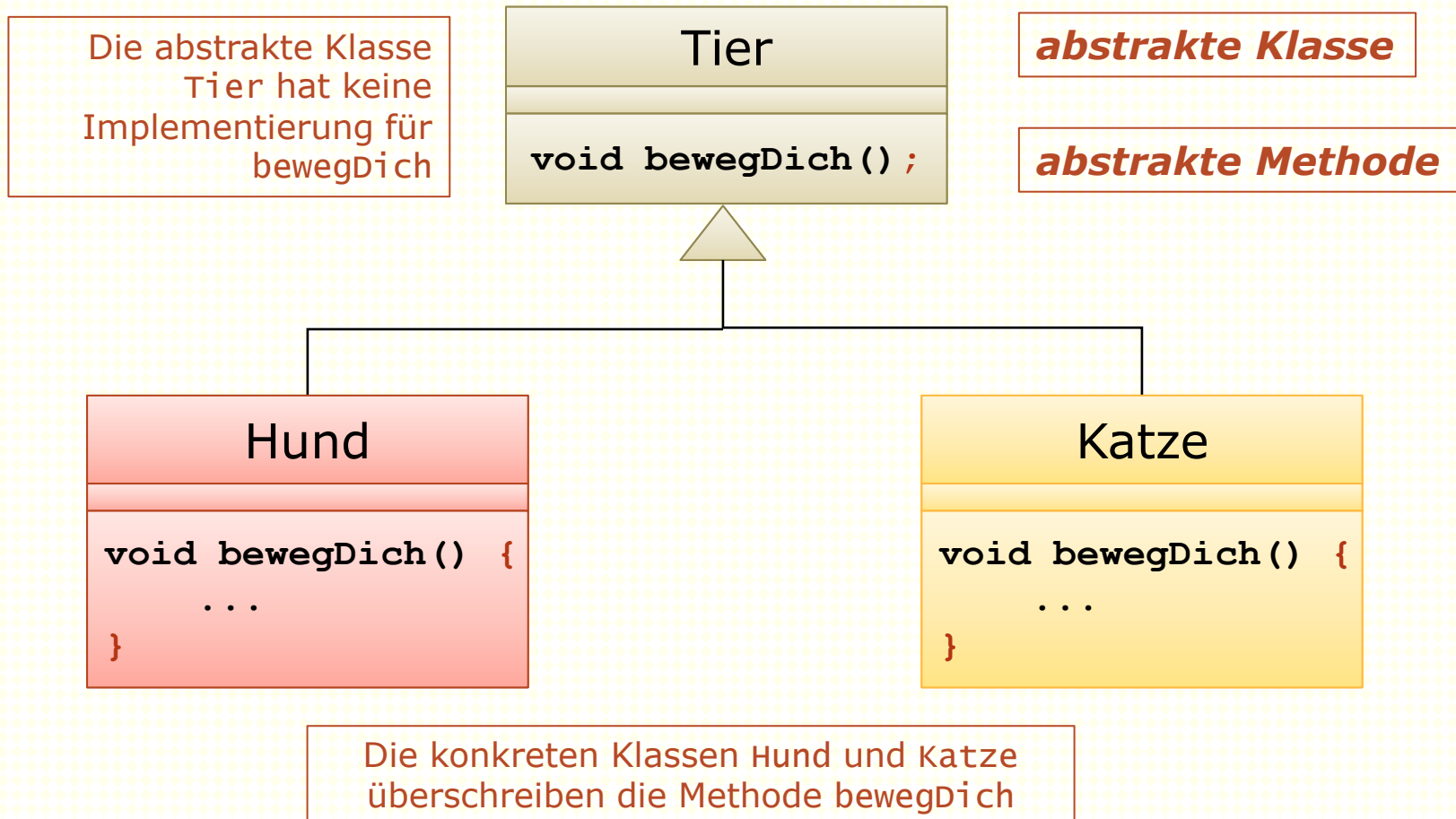
- **static** Methoden
- **private** Methoden (Instanz oder static)
- mit **final** gekennzeichnete Instanzmethoden
 - Definieren einer Methode mit derselben Signatur führt zu Compiler Fehler

```
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();  
}
```

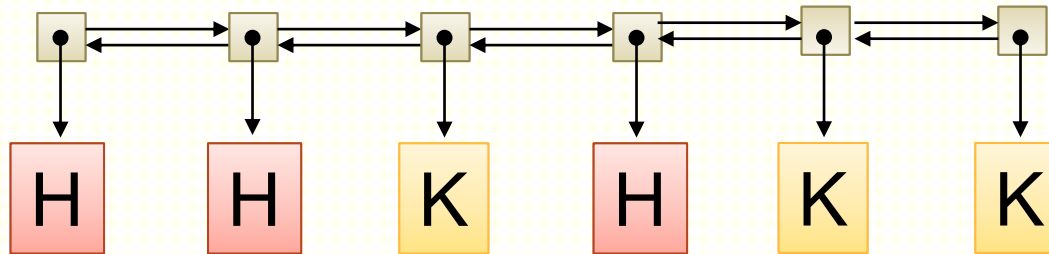
```
class Katze extends Tier {  
    @Override public void bewegDich() {  
        System.out.println("Katzenbuckel");  
    }  
}
```

Polymorphismus

▪ Abstrakte Klasse

- kann nicht instanziiert 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)

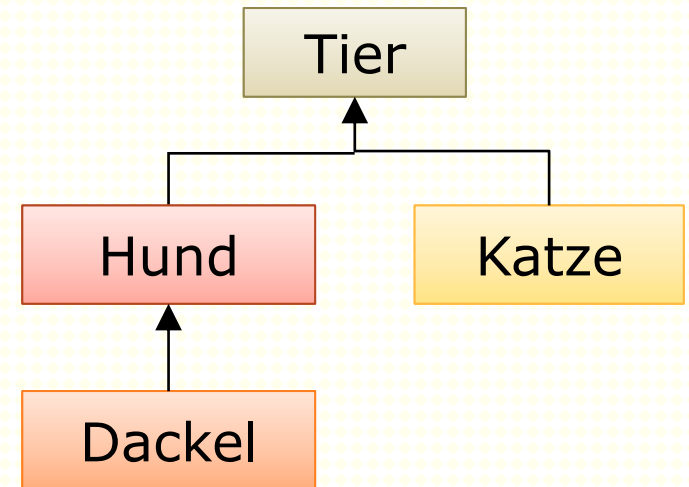
- **gilt seit Java 8 nicht mehr**

- 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();  
    }  
}
```

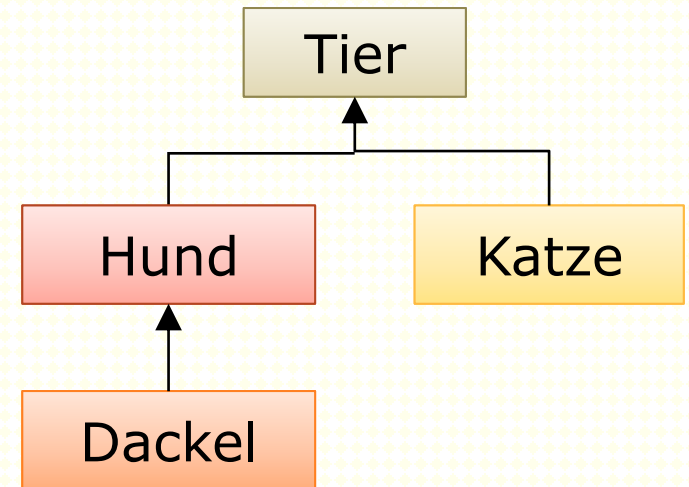
```
...  
tuWas(new Hund());  
tuWas(new Katze());  
tuWas(new Dackel());  
...
```

Typinformation

- **instanceof mit pattern matching**

- seit Java 14:
 - wenn das Objekt vom angegebenen Typ ist, wird es an die Variable gebunden
 - wenn nicht, steht die Variable nicht zur Verfügung

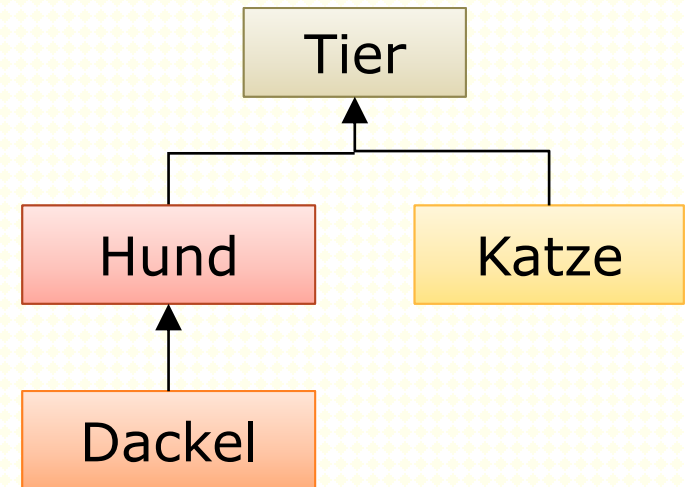
```
void tuWas(Tier t) {  
    // wenn t vom Typ Hund ist, wird  
    // das Objekt an h gebunden;  
    if (t instanceof Hund h) {  
        h.belle();  
    }  
}
```



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


▪ Typ-Objekt (Class)

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



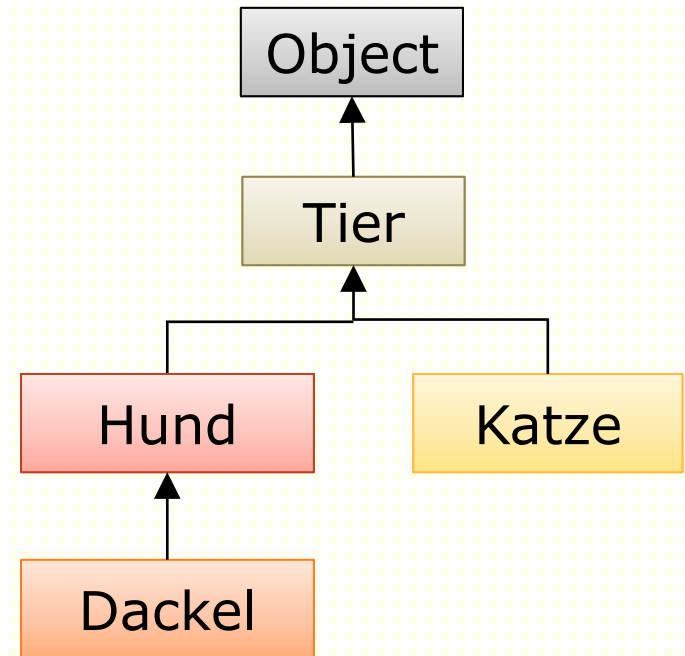
```
void tuWas(Tier t) {
    System.out.println(t.getClass().getName());
    // wenn es exakt der Typ Hund ist
    if (t.getClass() == Hund.class) {
        System.out.println("Exakt Typ Hund");
    }
}
```

```
...
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)

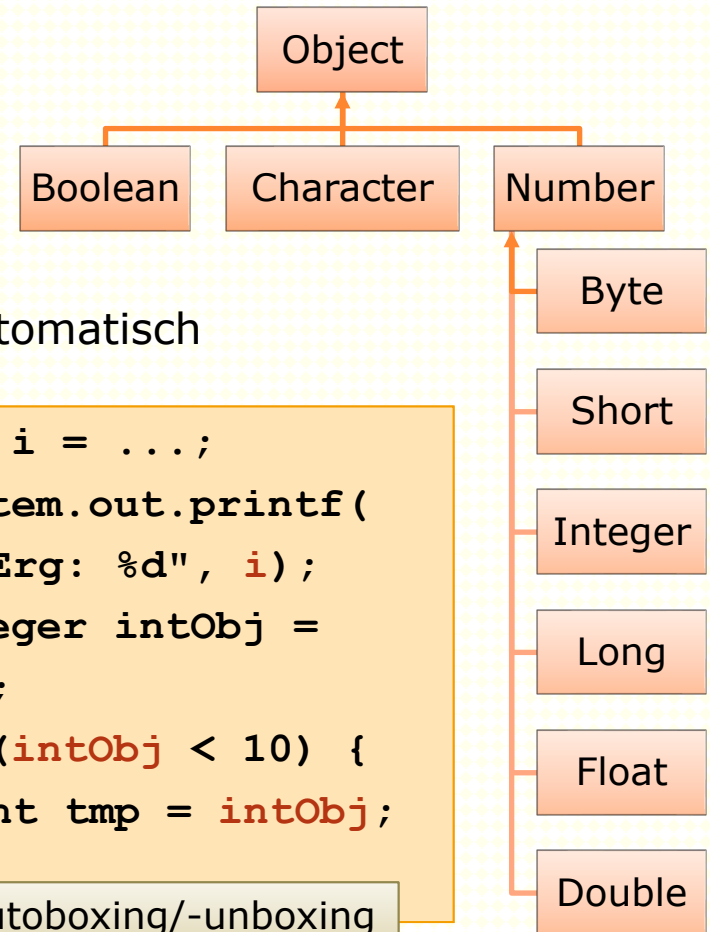
Achtung: `==` und `!=` vergleichen die Referenzen

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

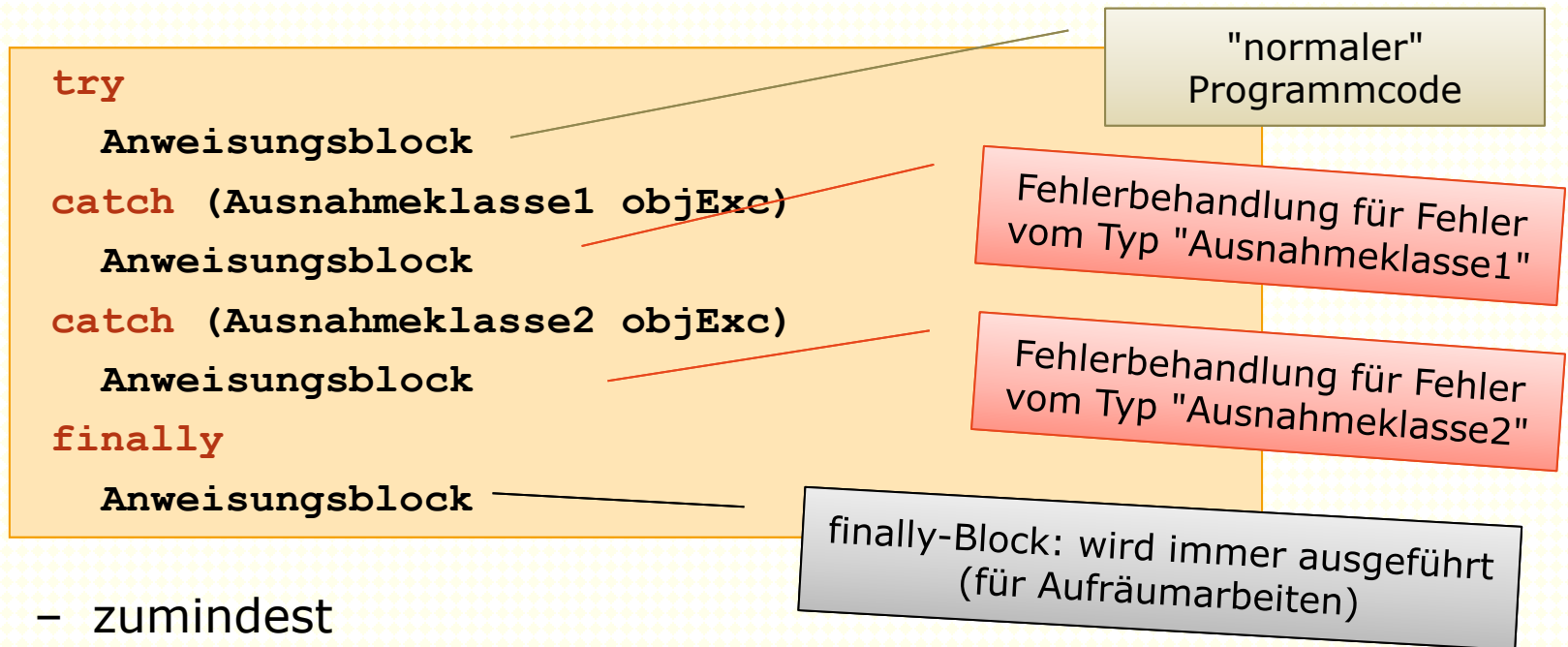
```
int i = ...;
System.out.printf(
    "Erg: %d", i );
Integer intObj =
    i;
if (intObj < 10) {
    int tmp = intObj;
}
```

Autoboxing/-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 (ArithmeticException e) {
    // Fehlerbehandlung für ArithmeticException
    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
ArithmeticException

Fehlerbehandlung für
NumberFormatException

finally-Block: wird immer
ausgeführt, falls vorhanden

Fehlerbehandlung mit Exceptions

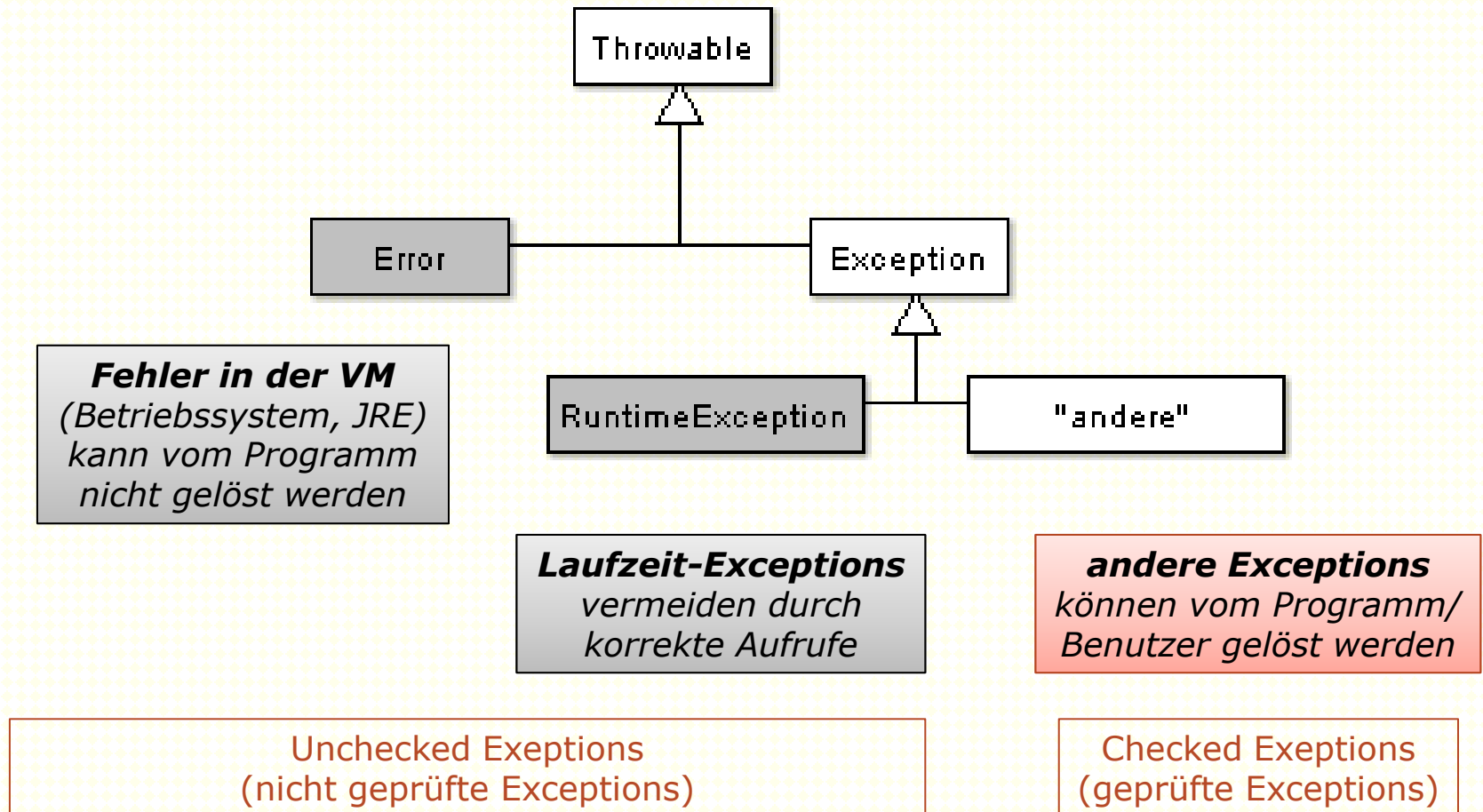
▪ Fehler auslösen

```
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

eigene Klasse als
Checked Exception

```
public class CalculationException extends Exception {  
    public CalculationException (String msg) {  
        super(msg);  
    }  
}
```

```
    public int calculate(char op, int z1, int z2)
```

```
        throws CalculationException {
```

die Exception mit
throws deklarieren

```
        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);
```

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 (ArithmeticException e) {
    // Fehlerbehandlung für ArithmeticException
    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

Annotationen

- **Metainformationen zu Klassen und ihren Member**
 - gehören nicht direkt zum Programm
 - haben keine direkte Auswirkung auf das Verhalten des betreffenden Codes
- **Verwendungszwecke**
 - Informationen für den Compiler
 - Informationen für Tools
 - z.B. zur automatisierten Code-Generierung
 - ...

Annotationen

■ Beispiele

- `@Deprecated`
 - eine Klasse oder Methode als veraltet kennzeichnen
- `@SuppressWarnings(value = "deprecated")` oder `@SuppressWarnings("deprecated")`
 - die Warnungen zu veralteten Methoden unterdrücken
- `@Override` - eine Methode als Überschreibung kennzeichnen (siehe Vererbung)

■ Syntax

- beginnen mit dem Zeichen `@`
- können Werte für Elemente enthalten
- wenn die Annotation 1 Element namens "value" hat, kann der Name entfallen

Annotationen

▪ Selber definieren mit @interface

- @Target gibt an, wo die Annotation stehen darf, z.B.
 - TYPE, METHOD, PARAMETER, ...
- @Retention gibt den Speicherort der Annotation an:
 - SOURCE, CLASS, RUNTIME
- @Repeatable: die Annotation kann mehrfach vorkommen
- @Inherited: die Annotation kann vererbt werden

```
@Target(TYPE)
```

```
public @interface Author {  
    String name();  
    String email();  
    String date();  
}
```

```
@Author (name = "Michaela",  
        email = "mp@mit.at",  
        date = "2021-07-31")  
public class Person {  
}
```

▪ 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 (Typlöschung)**

- Umsetzung in Java erfolgt mit Typlöschung: 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 kleiner als o2
0	Instanz und o2 sind gleich
positiv	Instanz größer als o2

Standardinterfaces: Sortierreihenfolge

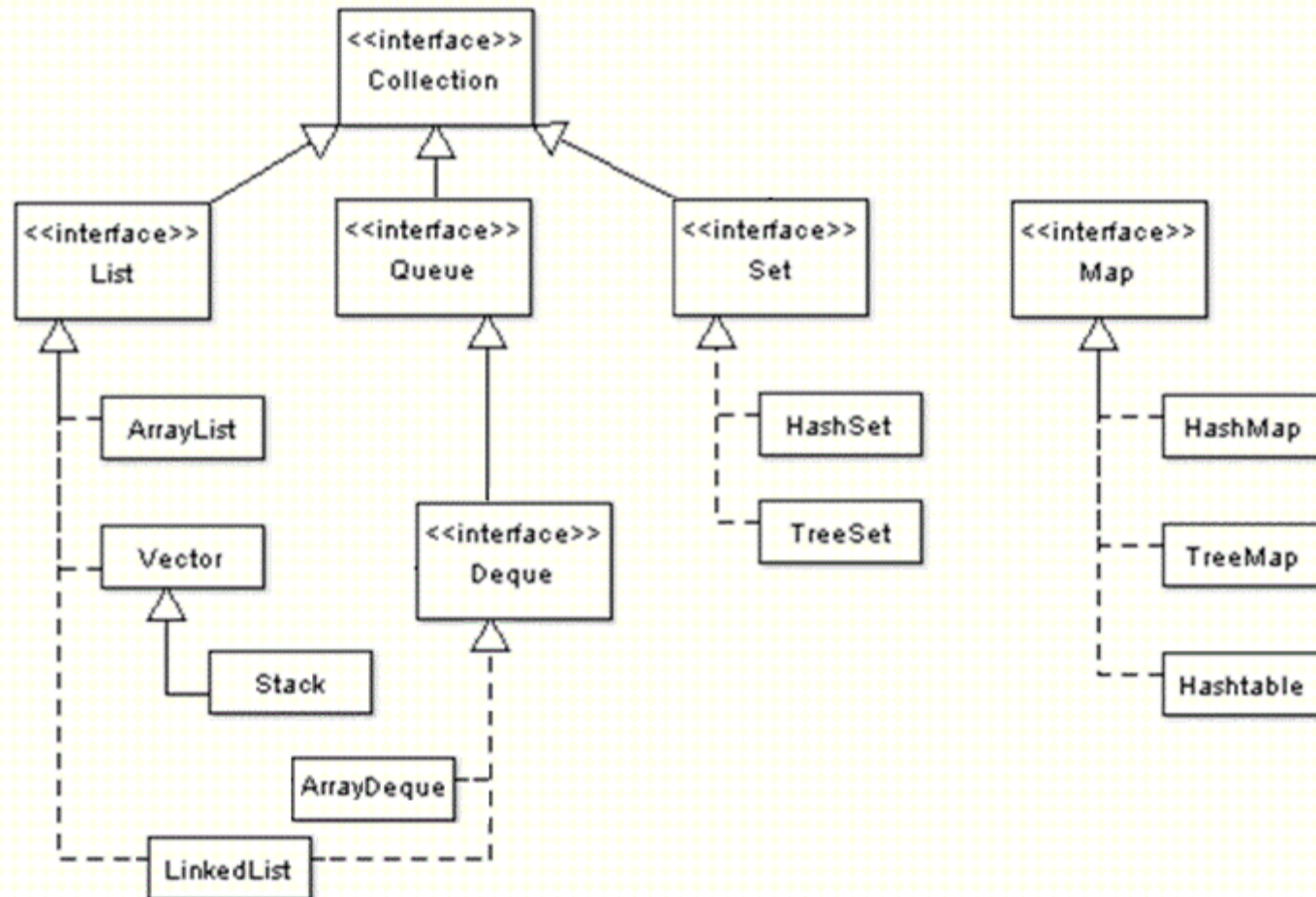
▪ **Comparator<T>**

- 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 kleiner als o2
0	o1 und o2 sind gleich
positiv	o1 größer als o2

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

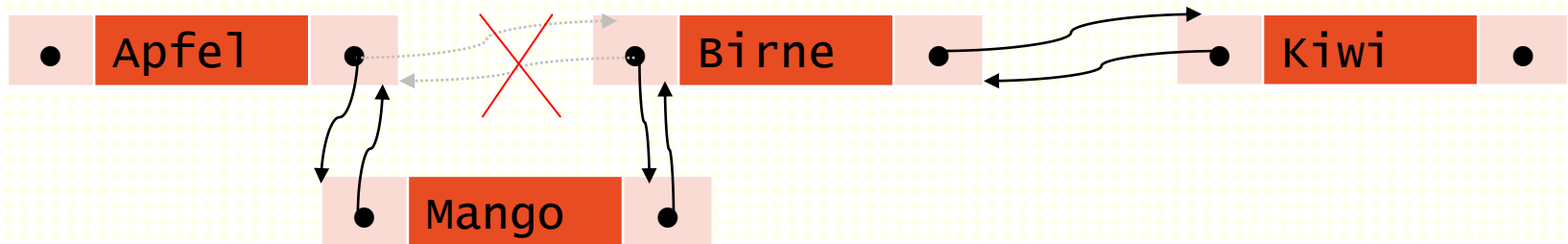
- **Verwaltet die Elemente mit verketteten Knoten**

- jeder Knoten hat potenziell 2 Nachbarn:
 - einen Vorgänger
 - einen Nachfolger
- rasche Einfüge- und Löschooperationen



1. Element

letztes Element



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

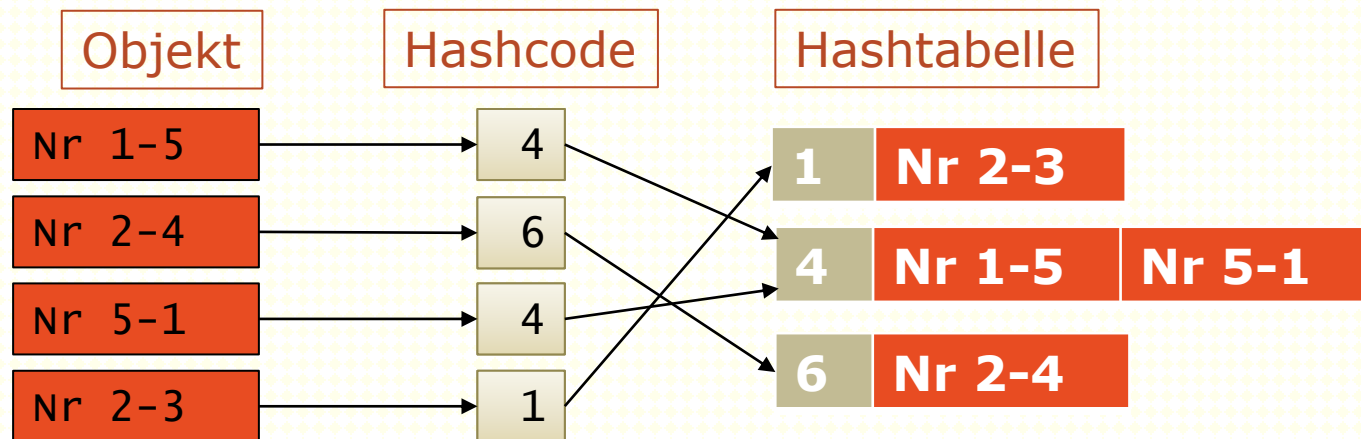
- HashMap
 - verwaltet die Paare nach dem Hashcode der Key-Elemente in **Hashtabellen**
 - **ungeordnet**
- TreeMap
 - verwaltet die Paare in einem **binären Suchbaum**
 - **sortiert** nach den Key-Elementen
- Hashtable
 - wie HashMap, mit **synchronized** methods

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, number;  
    @Override  
    public String toString() {  
        return abteilung  
            + "-" + number;  
    }  
    ...  
}
```



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;  
    }  
}
```

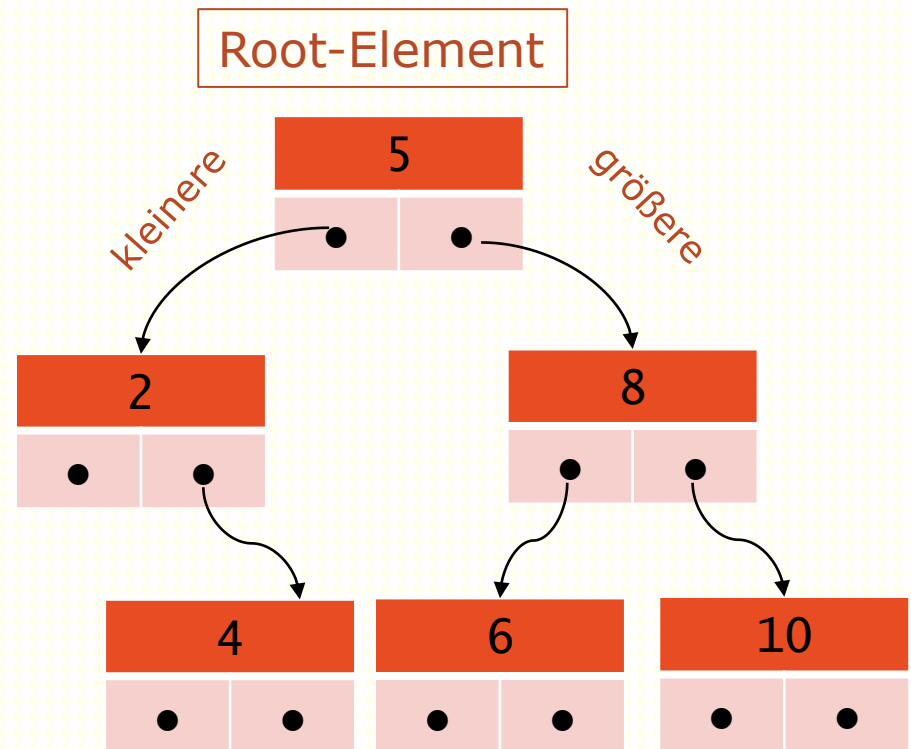
HashCode für das
Objekt liefern

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**

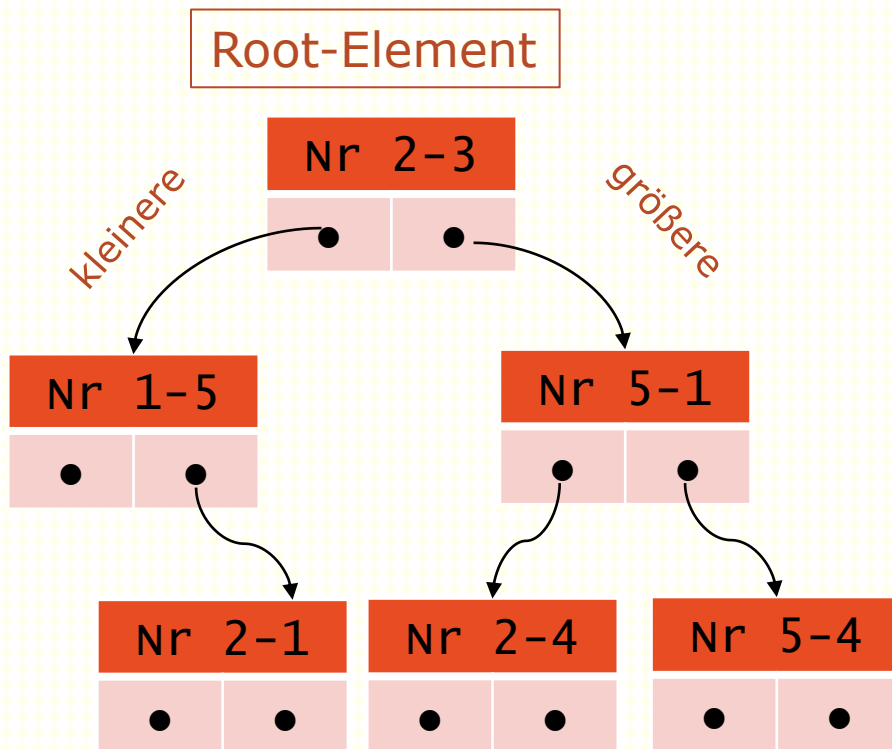
- über die `Comparable<E>`-Implementierung des Element-Typs
 - natürliche Sortierreihenfolge
- oder mit einem eigenen `Comparator<E>`
 - wird bei der Erzeugung angegeben

```
// 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

; 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>::<Methodenname>
```

Syntax	Art
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 Primitives
 - 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 API

▪ 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
- `mapToInt(ToIntFunction<T>)` /
`mapToLong(ToLongFunction<T>)` /
`mapToDouble(ToDoubleFunction<T>):`
 - liefern Stream von primitiven Werten, die mit der Function aus den Quell-Elementen berechnet werden

Stream API

▪ 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()`
- `toArray()` :
 - liefert ein Object-Array mit den Elementen
- `toArray(IntFunction<T[]>)` :
 - liefert ein typisiertes Array mit den Elementen
 - das Array muss man dafür selbst erzeugen

```
Animal[] array = animalList.stream()  
    .filter(Animal::isHerbivore)  
    .toArray(Animal[]::new);
```

Konstruktor-Referenz für die Erzeugung des Arrays

▪ Terminal Operations

- `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)

```
// das Tier mit dem kleinsten Gewicht holen
Optional<Animal> min = animalList.stream()
    .min(Comparator.comparing(Animal::getWeight)) ;
```

Optionale Ergebnisse

▪ Klasse **Optional** <T>

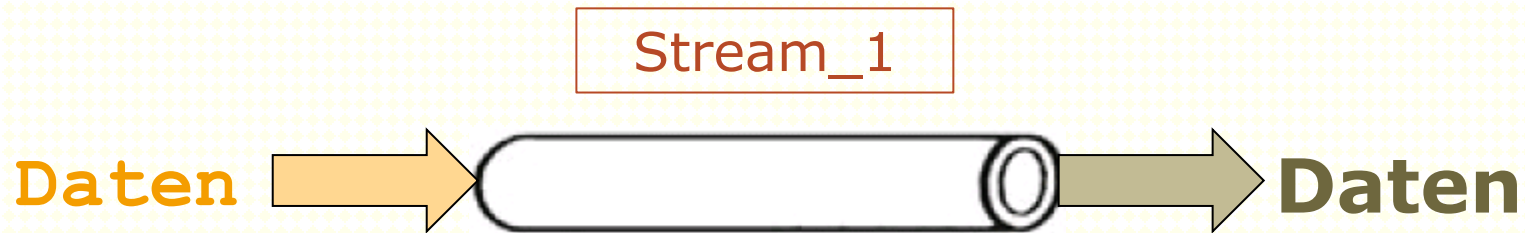
- Wrapper für einen Wert, der vorhanden sein kann oder auch nicht
 - vereinfacht das Handeln von null-References
 - wird von einigen Terminal Operations zurückgeliefert
- Methoden
 - `ifPresent`, `ifPresentOrElse`: einen Consumer ausführen, wenn ein Wert vorhanden ist
 - `isPresent`, `isEmpty`: Prüfen 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 angegebenen Wert erzeugen

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

Streams und FileIO

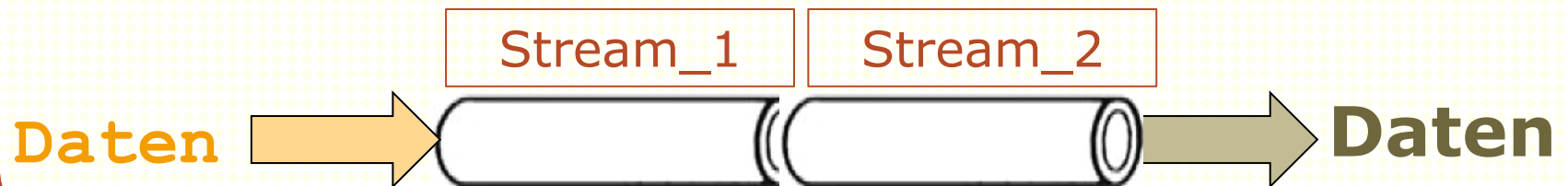
Streams

Streams transportieren Daten von A nach B ...



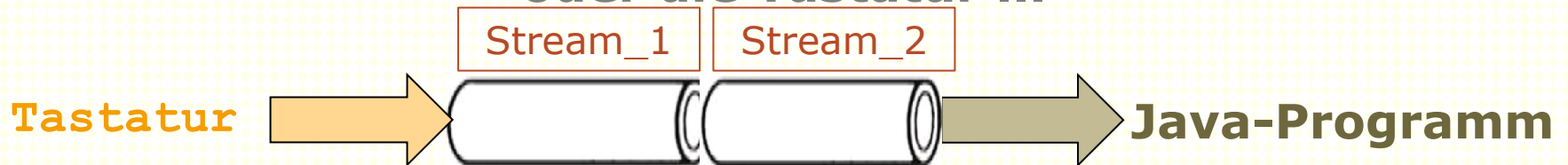
... und können die Daten dabei bearbeiten

Streams können aneinander gereiht werden

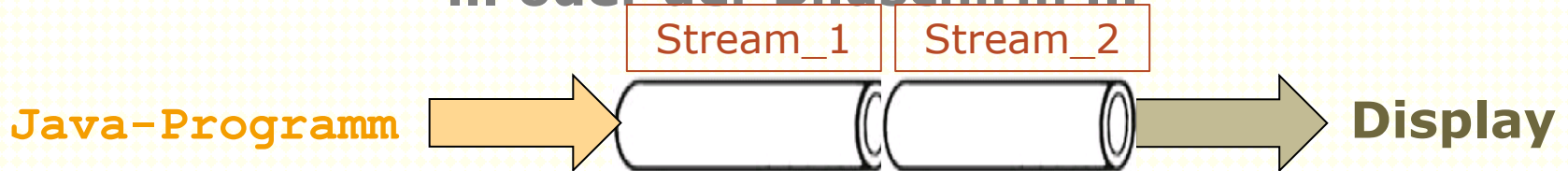


Streams

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

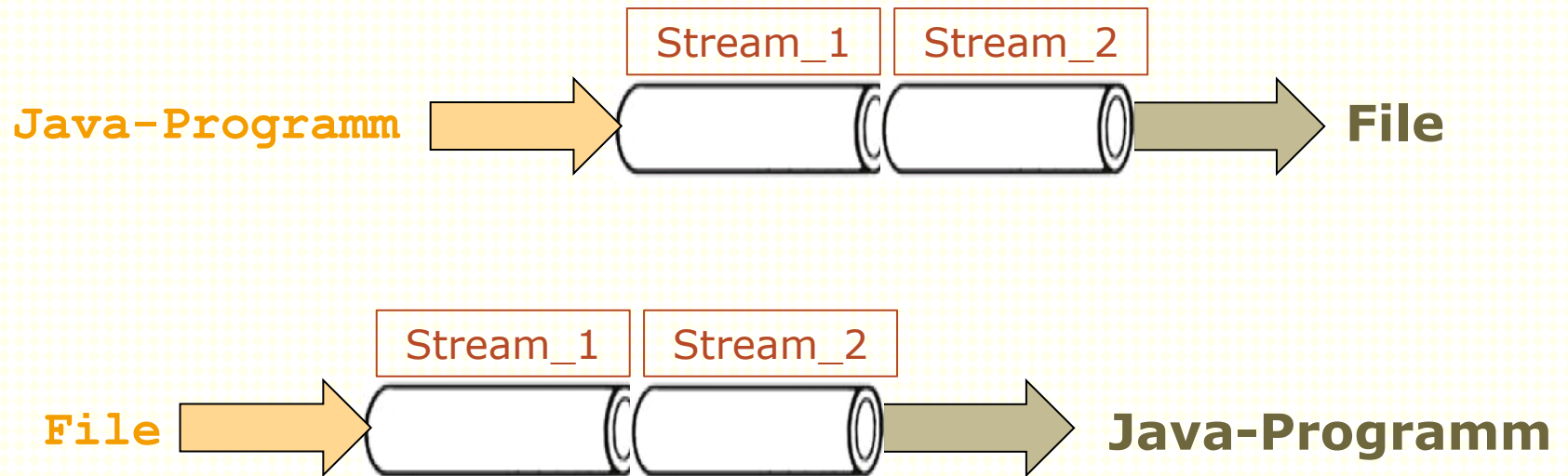


... oder der Bildschirm ...



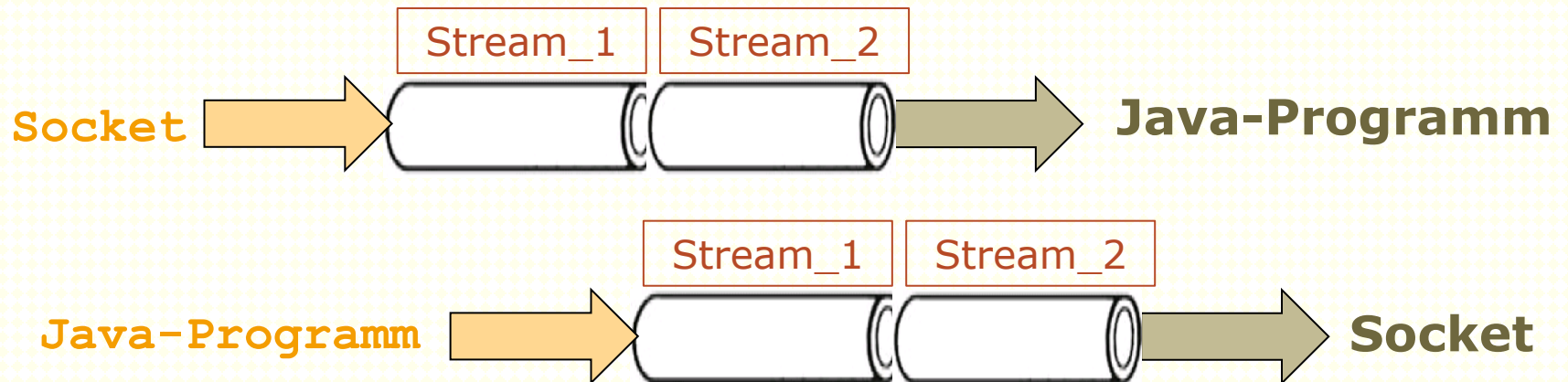
Streams

... oder ein File ...



Streams

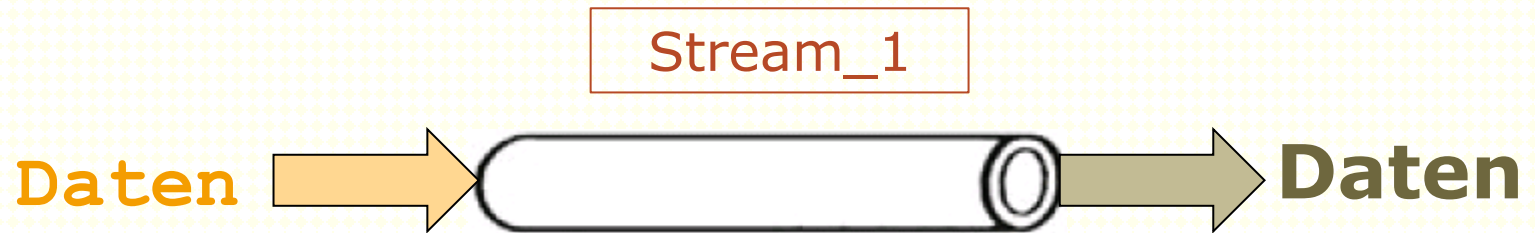
... oder das Ende einer Netzwerkverbindung
(Socket)



oder ...

Streams

Streams sind immer one-way



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

Arten von Streams

	Byte-Stream	Character-Stream
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, FileInputStream	FileWriter 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 FileReader	Text schreiben bzw. lesen an/von File	
PrintWriter	Primitive Daten als Text schreiben	print println printf
BufferedWriter	Zeilenweise schreiben	newline
BufferedReader	Zeilenweise lesen	readLine
StringWriter	An einen StringBuffer schreiben	getBuffer toString
StringReader	von einem String 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, FileOutputStream	Schreiben bzw. Lesen von / an Datei	
DataOutputStream	Primitive Daten binär schreiben	writeInt, writeChar, writeBoolean, writeUTF,
DataInputStream	Primitive Daten binär lesen	readInt, readChar, readBoolean, readUTF, skipBytes
ObjectOutputStream	Objekte schreiben (Serialisierung)	writeObject
ObjectInputStream	Objekte lesen (Serialisierung)	readObject

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 ObjectOutputStream

▪ **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)

▪ Unterstützung für Character Encodings

– Klasse **Charset**

- gibt Zugriff auf vordefinierte Character Encodings
- Charset-Instanzen können über ihren Namen abgerufen werden, z.B. für ANSI Kodierung (CP1252):

```
Charset cs = Charset.forName("windows-1252");
```

– Klasse **StandardCharsets**

- enthält Konstante für ein paar wichtige Charsets, z.B. UTF_8, US_ASCII, ISO_8859_1

```
Charset cs = StandardCharsets.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 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

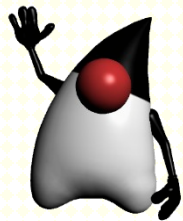
Hilfsklassen Path, Paths und Files (nio)

- **In java.nio wurde Funktionalität von File geteilt**
 - **Path** repräsentiert einen File- oder Verzeichnisnamen
 - **Paths**: Hilfsklasse zum Verketteten 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) { ... }
```

Java Unit Tests

Michaela Pum



OpenJDK



Unit Tests

- **Unit Tests (Modultests)**

- Testen einzelne Softwarekomponenten
- weisen nach, dass eine Komponente
 - technisch lauffähig ist
 - fachlich korrekt implementiert ist
- können im Zuge des Deployments automatisch ausgeführt werden

- **JUnit Test Framework**

- DAS Testframework für Java Unit Tests
- Testklassen werden mit Annotationen konfiguriert
- IDEs unterstützen Erstellen und Ausführen von Tests
- Build-Tools (z.B. Maven) unterstützen Ausführung während Deployment

JUnit 4 Annotationen

Annotation	Bedeutung
@Test	eine Methode als Unit-Testmethode kennzeichnen
@Ignore	eine Testmethode bei der Ausführung ausschließen
@BeforeClass / @AfterClass	Methode, die vor / nach allen Tests der Klasse ausgeführt wird
@Before / @After	Methode, die vor / nach jedem Test der Klasse ausgeführt wird
@FixMethodOrder	Reihenfolge der Testmethoden festlegen (z.B. nach Name aufsteigend)

JUnit 4 Assert

Methoden	Bedeutung
assertSame / assertNotSame	Prüft, ob ein Wert gleich / nicht gleich dem erwarteten Wert ist (mit Vergleichsoperatoren)
assertEquals / assertNotEquals	Prüft, ob ein Wert gleich / nicht gleich dem erwarteten Wert ist (mit equals)
assertArrayEquals	Prüft, ob ein Arrays den erwarteten Inhalt hat
assertTrue / assertFalse	Prüft, ob eine Bedingung zutrifft / nicht zutrifft
assertNull / assertNotNull	Prüft, ob eine Referenz null / nicht null ist
assertThrows	Prüft, ob eine bestimmte Exception auftritt
fail	Löst einen Fehler aus

Wenn die Prüfung fehlschlägt, wird eine Exception ausgelöst,
und der Test ist fehlgeschlagen

JUnit 4 Beispiel

```
public class BankAccountTest {  
    @Test  
    public void testWithdraw() throws BankException {  
        BankAccount acct1 = new BankAccount("Max", 1000);  
        acct1.withdraw(100);  
        Assert.assertEquals(-100, acct1.getBalance(), 0);  
    }  
  
    @Test  
    public void testWithdraw_amount_too_higt()  
        throws BankException {  
        BankAccount acct1 = new BankAccount("Max", 1000);  
        Assert.assertThrows(BankException.class,  
            () -> { acct1.withdraw(1001); });  
    }  
}
```

JUnit 5 Annotationen

Annotation	Bedeutung
@Test	Methode als Testmethode kennzeichnen
@BeforeAll / @AfterAll	Methode, die vor / nach allen Tests der Klasse ausgeführt wird
@BeforeEach / @AfterEach	Methode, die vor / nach jedem Test der Klasse ausgeführt wird
@Nested	Eingebettete Testklasse
@TestClassOrder	Reihenfolge der eingebetteten Testklassen festlegen (z.B. nach Name aufsteigend)
@TestMethodOrder	Reihenfolge der Testmethoden festlegen (z.B. nach Name aufsteigend)
@DisplayName	angezeigten Namen anpassen
@TestInstance	Lifecycle einer Testklasse festlegen (PER_METHOD oder PER_CLASS)

JUnit 5 Assert

Methoden	Bedeutung
assertIterableEquals	Prüft, ob ein Iterable die erwarteten Werte enthält
assertLinesMatch	Prüft, ob eine Reihe von Strings den erwarteten Werten entsprechen
assertInstanceOf	Prüft, ob eine Referenz vom erwarteten Typ ist
assertDoesNotThrow / assertAll	Prüft, ob ein / alle Executables ohne Exception ausgeführt werden
assertThrowsExactly	Prüft, ob eine bestimmte Exception auftritt (genauer Typ)
assertTimeout	Prüft, ob ein Executable innerhalb des erwarteten Intervalls ausgeführt wird

Außerdem stehen die meisten assert-Methoden
aus JUnit 4 weiterhin zur Verfügung

JUnit 5 Beispiel

```
class BankAccountTest {
    @Nested class BankAccountTransactionsTest{
        @Test
        public void testWithdraw() throws BankException {
            BankAccount acct1 = new BankAccount("Max", 1000);
            acct1.withdraw(100);
            Assertions.assertEquals(-100, acct1.getBalance(), 0);
        }
        @Test
        public void testWithdraw_amount_too_high()
            throws BankException {
            BankAccount acct1 = new BankAccount("Max", 1000);
            Assertions.assertThrowsExactly(BankException.class,
                () -> { acct1.withdraw(1001); });
        }
    }
}
```

JUnit 5 Conditions

- **@Disabled**
 - Testmethode bei der Ausführung ausschließen
- **@EnabledOnOs / @DisabledOnOs**
 - Ausführen in Abhängigkeit des Betriebssystems
- **@EnabledOnJre / @DisabledOnJre / @EnabledForJreRange / @DisabledForJreRange**
 - Ausführen in Abhängigkeit der Java-Version
- **@EnabledIfSystemProperty / @DisabledIfSystemProperty**
 - Ausführen in Abhängigkeit einer System-Property
- **@EnabledIfEnvironmentVariable / @DisabledIfEnvironmentVariable**
 - Ausführen in Abhängigkeit einer Umgebungsvariable

```
@Test @EnabledOnOs ({ OS.LINUX, OS.MAC })  
void testOnLinuxAndMac() { ... }  
  
@Test @EnabledForJreRange (min = JAVA_9)  
void testOnJava9toCurrent () { ... }
```

JUnit 5 Dependency Injection

▪ **ParameterResolver**

- Interface das die Injection von Objekten in Konstruktoren und Methoden ermöglicht
- vordefinierte Resolver für:
 - `TestInfo`: Information zum aktuellen Test (Klasse, Methode, Anzeigename)
 - `RepetitionInfo`: bei wiederholten Tests Informationen zur Wiederholung
 - `TestReporter`: Objekt über das aus einem Test Daten publiziert werden können (z.B. einfache Zeichenfolge oder Name-Value-Paar)
- eigene Resolver können mit per Extension-Klasse erstellt und eingebunden werden

JUnit 5 weiterführende Techniken

▪ @RepeatedTest

- Testmethode wiederholt ausführen

```
@RepeatedTest(value = 5)
void testAutoIncrementNumbers(RepetitionInfo ri) {
    System.out.printf("Test call %d of %d\n",
        ri.getCurrentRepetition(), ri.getTotalRepetitions());
    ...
}
```

▪ @ParameterizedTest

- Testmethode mit Parametern

```
@ParameterizedTest
@ValueSource(doubles = { -1, 15001, 100_000 })
void test_Saving_Deposit_Exception(double amount) {
    System.out.printf("Test with amount=%.2f\n", amount);
    ...
}
```

JUnit 5 Werte für @ParameterizedTest

Annotation	Bedeutung
@ValueSource	Werte stehen in einem Array von Konstanten
@NullSource / @EmptySource	Mit null oder einem leeren Argument testen
@EnumSource	Werte sind die Instanzen eines Enums
@MethodSource	Eine Methode stellt die Werte als Stream<T> bereit
Arguments	fasst Werte zusammen, wenn die Testmethode mehrere Parameter hat
Arguments Accessor	fasst mehrere Parameter zusammenfassen

JUnit 5 Werte für @ParameterizedTest

```
@ParameterizedTest
@MethodSource("checkingAccounts")
void testBankAccount_Checking(String name, double overdraw) {
    System.out.printf("test values %s, %f\n", name, overdraw);
    ...
}

static Stream<Arguments> checkingAccounts() {
    return Stream.of(
        Arguments.of("Max", 0.0),
        Arguments.of("Moritz", 1000.0),
        Arguments.of("Pippi", 10000.0));
}
```

JUnit 5 Werte für @ParameterizedTest

Annotation	Bedeutung
@CsvSource	Werte stehen in CSV-Zeichenfolgen
@CsvFileSource	Werte stammen aus CSV-Datei am Classpath
@Arguments Source	Werte stammen von eigenem ArgumentsProvider

```
@ParameterizedTest
@CsvSource({ "Max, 0.3, 0", "Moritz, 0.25, 1000" })
void testBankAccount_Saving(ArgumentsAccessor args) {
    System.out.printf("Test with values %s - %.2f - %.2f\n",
        args.getString(0), args.getDouble(1), args.getDouble(2));
    ...
}
```


Mockito – Mock-Objekte in JUnit Tests

- **Mock(-Objekt)**

- Platzhalter für ein echtes Objekt während einem Unittest
- zum Testen der Interaktion eines anderen Objekts mit dem Mock

- **Mocking Framework**

- stellt solche Mocks für beliebige Klassen / Interfaces bereit
- vielfach werden dadurch Fake-Implementierungen für die Mock-Objekte unnötig

- **Mockito**

- weitverbreitetes Mocking Framework für Java
- implementiert eine Extension für JUnit 5
 - direkte Verwendung mit JUnit 5 möglich

Mockito – Mocks erzeugen

- **Mockito.mock**

- erzeugt einen Mock zu einer Klasse oder einem Interface

- **@Mock**

- Attribut oder Parameter als Mock kennzeichnen
 - Testklasse muss mit @ExtendWith(MockitoExtension.class) gekennzeichnet sein

```
@ExtendWith(MockitoExtension.class) class RepoTests{  
    @Test void test_add_account() {  
        BankAccount mockAcct = Mockito.mock(BankAccount.class);  
        ...  
    }  
    @Test void test_remove_account(@Mock BankAccount mock) {  
        ...  
    }  
}
```

Mockito – Mocks konfigurieren I

- **when(...).thenReturn(...)**
 - festlegen, dass beim Aufruf einer Methode des Mock-Objekts ein bestimmter Wert zurückgeliefert wird
- **when(...).thenThrow(...)**
 - festlegen, dass beim Aufruf einer Methode des Mock-Objekts eine bestimmter Exception geworfen wird

```
// unser Fake-Objekt soll 99 als accountNumber liefern
when(mockAcct.getAccountNumber()).thenReturn(99);
// unser Fake-Objekt soll beim Aufruf von getMaxOverdraw
// eine IllegalStateException werfen
when(mockAcct.getMaxOverdraw()).
    thenThrow(IllegalStateException.class);
// beim 1. Mal von getBalance 14, dann 9 liefern
when(acct.getBalance()).thenReturn(14.0).thenReturn(9.0);
when(acct.getBalance()).thenReturn(14.0, 9.0);
```

Mockito – Mocks konfigurieren II

- **doReturn(...).when(...)**
 - festlegen, dass beim Aufruf einer Methode des Mock-Objekts ein bestimmter Wert zurückgeliefert wird
- **doThrow (...).when(...)**
 - festlegen, dass beim Aufruf einer Methode des Mock-Objekts eine bestimmter Exception geworfen wird
 - auch für void-Methoden

```
// unser Fake-Objekt soll 99 als accountNumber liefern
doReturn(mockAcct).when(repo).get(99);
// unser Fake-Objekt soll beim Aufruf von deposit
// in jedem Fall eine BankException werfen
doThrow(BankException.class)
    .when(mockAcct).deposit(anyDouble());
```

Mockito – Spy

- **Spy**
 - ist eine Hülle für ein echtes Objekt
 - Zeichnet Interaktionen mit dem Objekt auf
 - Methoden werden an das echte Objekt delegiert, sofern sie nicht extra konfiguriert wurden
- **Mockito.spy(...)**
 - erzeugt einen Spy zu einem echten Objekt
- **@ Spy**
 - kennzeichnet ein Feld als Spy für das Objekt

```
BankAccount acct = spy(new BankAccount("X", 1000));  
// als AccountNumber unseres Kontos 99 liefern  
when(mockAcct.getAccountNumber()).thenReturn(99);  
assertEquals(99, mockAcct.getAccountNumber());  
assertThrows(BankException.class, () -> acct.deposit(15001));
```

Mockito – verify

▪ Verhaltenstest

- prüft ob ein erwartetes Verhalten aufgetreten ist, z.B. ob
 - Methode (mit bestimmten Argumenten) aufgerufen wurde
 - nicht aufgerufen wurde
 - genau / mindestens / höchstens x Mal aufgerufen wurde
 - ...
- Argumente können per Wert oder ArgumentMatcher angegeben werden

```
verify(mockAcct).deposit(15001); // genau 1x
verify(acct, never()).withdraw(anyDouble()); // nie
verify(acct, times(1)).deposit(anyDouble()); // genau 1x
verify(acct, atLeast(1)).getAccountNumber(); // mindestens 1x
verify(acct, atMostOnce()).getMaxOverdraw(); // höchstens 1x
verifyNoMoreInteractions(acct); // keine weiteren Aufrufe
```

Mockito – Dependency Injection

- **@InjectMocks**

- Dependency Injection von Mock-Objekten

```
@ExtendWith(MockitoExtension.class)
public class BankServiceTests {
    @Mock BankRepository repo;

    @InjectMocks BankService svc;

    @Test
    void test_injected() {
        assertNotNull(svc);
    }
}
```