Programmieren 1 (Java)

Zusammenfassung v1.2

Kälin Thomas, Abteilung I
WS 05/06

1. **Darstellungstechniken** 6
1.1. **Nassi-Shneiderman-Diagramme** 6
1.1.1. Sequenz 6
1.1.2. Selektion (If..Then / If..Then..Else) 6
1.1.3. Mehrfach Alternative (Switch) 6
1.1.4. Iterationen 6
1.2. **UML-Diagramme** 7
1.2.1. Klasse / Objekt 7
1.2.2. Aggregation 7
1.2.3. Vererbung 7
1.2.4. Interfaces 7
1.2.5. Pakete 7
2. **Grundlegendes zu Java** 8
2.1. **Eigenschaften von Java** 8
2.2. **Virtuelle Maschine** 8
2.3. **Programmerzeugung und Ausführung** 8
2.3.1. Lexikalische Analyse 8
2.3.2. Syntaxanalyse 8
2.3.3. Semantische Analyse 8
2.3.4. Codeerzeugung 8
2.3.5. Virtuelle Maschine 8
3. **Lexikalische Konventionen** 9
3.1. **Unicode** 9
3.2. **ASCII-Tabelle** 9
3.3. **Lexikalische Einheiten** 9
3.3.1. Kommentare 9
3.3.2. Javadoc 9
3.4. **Styleguide** 9
3.5. **Schlüsselwörter in Java** 9
3.6. **Konstanten** 9
3.6.1. Literale Konstanten 9
3.6.2. Symbolische Konstanten 10
3.7. **Zeichenkonstanten** 10
3.8. **Operatoren** 10
4. **Datentypen und Variablen** 11
4.1. **Einfache Datentypen** 11
4.1.1. Ganzzahlig 11
4.1.2. Gleitpunkttypen (nur a/2^n exakt darstellbar!) 11
4.1.3. Logisch 11
4.1.4. Zweierkomplement 11
4.1.5. Gleitpunkttypen (umrechnen) 11
4.2. **Referenztypen** 12
4.2.1. Referenzen 12
4.2.2. Array 12
4.2.3. Enum 12
4.3. **Variabeltypen** 12
4.3.1. Klassenvariable (static) 12
4.3.2. Instanzvariable 12
4.3.3. Lokale Variablen 12
4.3.4. Konstanten (final) 12
4.4. **String-Klassen** 12
4.4.1. String 12
4.4.2. StringBuffer / StringBuilder 13
4.5. **Wrapper-Klassen** 13
4.6. **Boxing** 13
4.7. **Zugriffsmodifikatoren** 13
5. **Ausdrücke und Operatoren (200)** 14
5.1. **Operatoren** 14

5.2. Ausdruck / Anweisung 14
5.3. Nebeneffekte 14
5.3.1. Nebeneffekt von Operatoren 14
5.3.2. Nebeneffekt von Methoden 14
5.4. Auswertungsreihenfolge 14
5.4.1. Assoziativität 14
5.4.2. Fieser Programmierer 14
5.5. Sonderfälle 14
5.6. Bitoperatoren 14
5.7. Bit-Shift-Operatoren 15
5.7.1. Linksshift-Operator 15
5.7.2. Vorzeichenbehafteter Rechtsshift-Operator 15
5.7.3. Vorzeichenloser Rechtsshift-Operator 15
5.8. Casten 15
5.8.1. Implizites Casten 15
5.8.2. Explizites Casten 15
6. Kontrollstrukturen 16
6.1. Block 16
6.1.1. Verschachtelte Blöcke 16
6.2. If..Then..Else 16
6.3. Switch 16
6.4. Schleifen / Iterationen 16
6.4.1. While 16
6.4.2. Do.. While 16
6.4.3. For 16
6.4.4. Endlosschleifen 17
6.4.5. For each 17
6.5. Break 17
6.6. Continue 17
6.7. Sprungmarken 17
7. Blöcke und Methoden 18
7.1. Gültigkeit / Sichtbarkeit 18
7.1.1. Beispiel: Sichtbarkeit 18
7.2. Methoden 18
7.2.1. Rückgabewerte 18
7.2.2. Finale Methoden 18
7.3. Polymorphie 18
7.4. Überladen von Methoden 18
7.5. Variabel lange Parameterliste 18
8. Klassen und Objekte 19
8.1. Allgemeines 19
8.1.1. Definition einer Klasse 19
8.1.2. Instantierung 19
8.1.3. Information Hiding 19
8.1.4. Identifikation einer Klasse 19
8.2. This 19
8.3. Initialisierung 19
8.3.1. Defaultwerte für Datentypen 19
8.3.2. Initialisierungsblock 19
8.4. Konstruktoren 20
8.5. Verhindern der Instantierung (Singleton) 20
8.6. Freigabe von Speicher 20
8.6.1. Garbage Collector 20
8.6.2. Finalize 20
8.7. Spezielle Klassen 21
8.7.1. Object 21
8.7.2. Class 21
8.8. Finale Klassen 21
8.9. Abstrakte Klassen 21

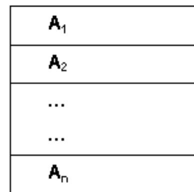
8.10. Generische Klassen **21**
9. Vererbung und Polymorphie **22**
9.1. Vererbung **22**
9.1.1. Definition 22
9.1.2. Verdeckte Datenfelder 22
9.1.3. Überschreiben von Methoden 22
9.1.4. Beispiel 22
9.1.5. Konsistenzhaltung 22
9.2. Polymorphie **22**
9.2.1. Definition 22
9.2.2. Object 23
9.2.3. Beispiel 23
9.2.4. Typenkonvertierung 23
10. Pakete **24**
10.1. Das Wichtigste **24**
10.2. Deklaration **24**
10.3. Import von Paketen **24**
11. Exception Handling **25**
11.1. Grundlagen **25**
11.1.1. Checked Exception 25
11.1.2. Unchecked Exception 25
11.2. Try/Catch **25**
11.3. Propagieren von Exceptions **25**
11.4. Vorteile **25**
12. Schnittstellen **26**
12.1. Interface / Implements **26**
12.2. Vererbung von Schnitstellen **26**
12.3. Probleme mit Schnittstellen **26**
12.3.1. Gleichnamige Konstanten 26
12.3.2. Exakt gleiche Methoden 26
12.3.3. Gleiche Methoden mit unterschiedlichen Exceptions 26
12.3.4. Unterschiedliche Rückgabewerte bei Methoden 26
12.4. Unterschiede: Abstrakte Klasse / Schnittstelle **26**
12.5. Spezielle Interfaces **26**
12.5.1. Comparable 26
12.5.2. Cloneable 27
12.5.3. Serializable 27
13. Ein-/Ausgabe und Streams **28**
13.1. Klassifizierung **28**
13.1.1. Sink-, Spring- und Processingstreams 28
13.1.2. Read-Funktion: int-Wert als Rückgabe? 28
13.2. Codebeispiele **28**
13.2.1. FileReader / FileWriter 28
13.2.2. ObjectInputStream 28
14. Collections **29**
14.1. Überblick **29**
14.1.1. Listen 29
14.1.2. Queue 29
14.1.3. Menge / Set 29
14.1.4. Assoziative Arrays / Maps 30
14.2. Iterator **30**
14.2.1. Codebeispiel 30
15. Sonstiges **31**
15.1. printf() **31**
15.2. Geschachtelte Klassen **31**
15.3. Applets **31**
15.4. Swing **31**
15.4.1. Java GUI 31
15.4.2. Panes 31

15.4.3. Komponenten 32
15.4.4. Container 32
15.4.5. Menüs 32
15.4.6. Sonstige Komponenten 32
15.4.7. Layoutmanager 32
15.4.8. Verarbeitung von Ereignissen 32
15.4.9. Delegations-Modell 32
16. Anhang 33
16.1. Schlüsselwörter (Beispiele) 33
16.2. Operatoren (Beispiele) 33
16.3. Sichtbarkeit 34
16.3.1. Sichtbarkeit und Vererbung 34

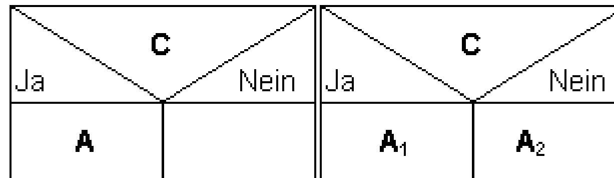
1. Darstellungstechniken

1.1. Nassi-Shneiderman-Diagramme

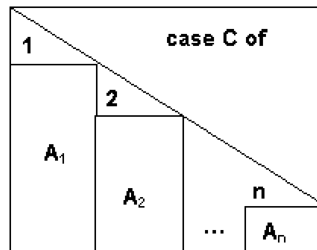
1.1.1. Sequenz



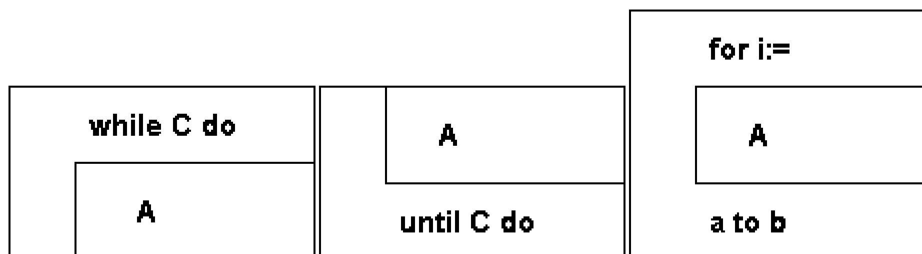
1.1.2. Selektion (If..Then / If..Then..Else)



1.1.3. Mehrfach Alternative (Switch)

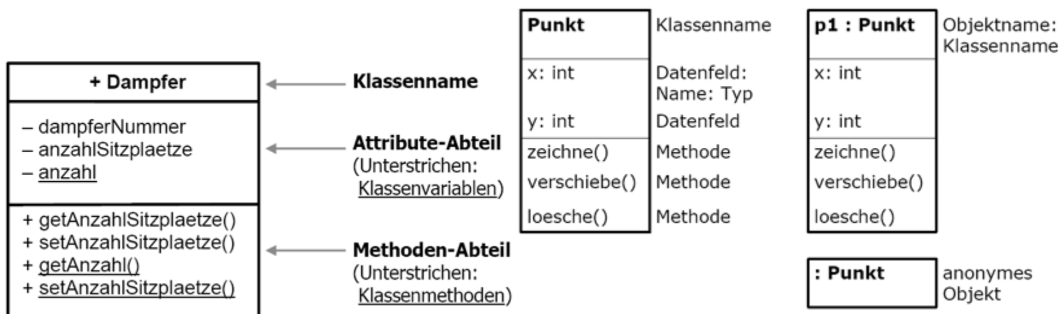


1.1.4. Iterationen

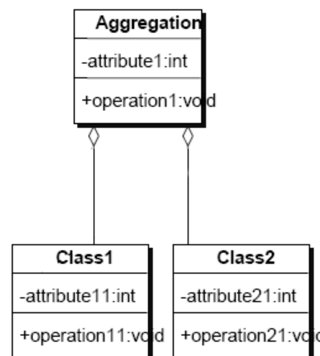


1.2. UML-Diagramme

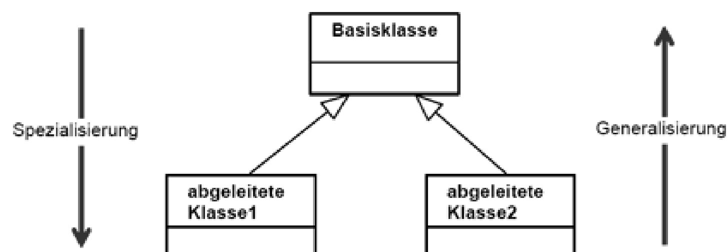
1.2.1. Klasse / Objekt



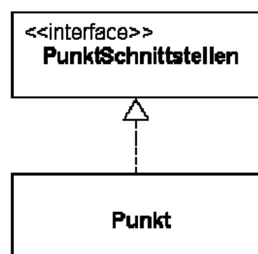
1.2.2. Aggregation



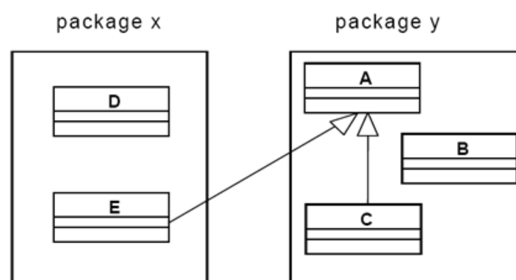
1.2.3. Vererbung



1.2.4. Interfaces



1.2.5. Pakete



2. Grundlegendes zu Java

2.1. Eigenschaften von Java

- Einfachheit und Stabilität: Zur Erhöhung der Einfachheit der Programmiersprache und der Stabilität der Programme wurden in Java gegenüber C und C++ verschiedene Sprachkonstrukte (Zeiger) weggelassen.
- Objektorientiertheit: Mit Java wurde eine echte objektorientierte Sprache entworfen, die den Programmierer zwing, objektorientiert zu programmieren.
- Verteilbarkeit: Java wurde von Anfang an für die Verteilung von Programmen auf verschiedene Rechner entworfen. Infolge einer umfangreichen Unterstützung durch die Java-Klassenbibliothek ist Java nahezu optimal für die Client/Server-Programmierung.
- Sicherheit: Java hat wie kaum eine andere Sprache ein mehrstufiges Sicherheitskonzept, das die Ausführung von kritischen oder für das System gefährlichen Operationen verhindert.
- Portierbarkeit
  - Bytecode: Durch die Kompilierung in eine von der Rechnerplattform unabhängigen Bytecode, der von der virtuellen Maschine ausgeführt wird, ist Java unabhängig vom jeweiligen Betriebssystem und der zugehörigen Rechner-Hardware. Software-Entwickler müssen ihre Programme nicht für jede Rechner-Plattform speziell anpassen.

2.2. Virtuelle Maschine

Bei Java wird bei der Kompilierung aus dem Quellcode kein Maschinencode, sondern Bytecode erzeugt. Der Bytecode wird dann von einem Interpreter (Java Virtual Machine, JVM), die für jede Rechnerplattform angepasst werden muss, zur Ausführung gebracht.

2.3. Programmerzeugung und Ausführung

Anders als bei anderen Programmiersprachen, wird bei Java kein Maschinencode erzeugt. Aus der Quellcode-Datei wird durch den Compiler (javac) eine interpretierbare .class-Datei erzeugt.

2.3.1. Lexikalische Analyse

Bei der lexikalischen Analyse wird versucht, in der Folge der Zeichen eines Programmes Wörter der Sprache zu erkennen.

2.3.2. Syntaxanalyse

Im Rahmen der Syntaxanalyse wird geprüft, ob die ermittelten Symbolfolgen zu der Menge der zulässigen Symbolfolgen gehören.

2.3.3. Semantische Analyse

Die semantische Analyse versucht die Bedeutung der Wörter herauszufinden. Neben der Überprüfung der Verwendung von Namen im Rahmen der Gültigkeitsbereiche spielt die Überprüfung von Typenverträglichkeit eine Hauptrolle.

2.3.4. Codeerzeugung

Es wird Bytecode erzeugt.

2.3.5. Virtuelle Maschine

Die VM übernimmt Aufgaben wie die Speicherverwaltung des Programms zur Laufzeit und ist für die Ein-/Ausgabe-Operationen und für Interaktion mit dem Betriebssystem zuständig.

3. Lexikalische Konventionen

3.1. Unicode

Java benutzt den Unicode-Zeichensatz. Java benutzt den UTF16-Code (16 Bit, 0..FFFF). Die ersten 128 Zeichen des Unicodes sind die Zeichen des 7-Bit ASCII-Zeichensatzes.

```
\u003F => Steht für das Zeichen „?“
0*16^3 + 0*16^2 + 3*16^1 + 15*16^0 = 63
```

3.2. ASCII-Tabelle

```
ASCII-Tabelle
    0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0  NUL SOH STX ETX EOT ENQ ACK BEL BS  HT  LF  VT  FF  CR  SO  SI
1  DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM  SUB ESC FS  GS  RS  US
2   SP  !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /
3   0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?
4   @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
5   P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _
6   `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o
7   p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~  DEL
```

3.3. Lexikalische Einheiten

- Namen, Schlüsselwörter, Konstanten, Satzzeichen ([] () {} ;), Operatoren
- Trennzeichen: Whitespaces, Kommentare

3.3.1. Kommentare

```
/* Ein Kommentar bis zum schliessenden Tag */

// Auch eine Zeile
```

3.3.2. Javadoc

```
/**
  * Ich bin ein Javadoc-Kommentar
  * @version 1.0
  * @author TK
  * @param intPara Blabla
  * @return Blabla
  */
```

- Umwandlung per javadoc –version -author DocuTest.java.

3.4. Styleguide

```
intVariable        -> Variable, klein
intMAXIMUM         -> Konstante, alles gross
getDrugs()         -> Methoden, klein
class Thomas {}    -> Klassen, Anfangsbuchstabe gross
```

3.5. Schlüsselwörter in Java

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

3.6. Konstanten

3.6.1. Literale Konstanten

```
3.14159
```

3.6.2. Symbolische Konstanten

```
public final double dblMAXIMUM = 10;
```

3.7. Zeichenkonstanten

\b	Backspace	BS(\u0008)
\t	horizontaler Tabulator	HT(\u0009)
\n	neue Zeile	LF(\u000a)
\f	neue Seite	FF(\u000c)
\r	Wagenrücklauf	CR(\u000d)
\"	"	(\u0022)
\'	'	(\u0027)
\\	\	(\u005c)

3.8. Operatoren

=	<	>	!	~	?:
==	!=	<=	>=	&&	||
++	--	+	-	*	/
&	|	^	%	<<	>>
>>>	+=	-=	*=	/=	&=
|=	^=	%=	<<=	>>=	>>>=

4. Datentypen und Variablen

4.1. Einfache Datentypen

4.1.1. Ganzzahlig

```
char    16bit-Unicode     0..65535 (kein Vorzeichenbit)
byte    8bit-Ganzzahl     -2^7 .. 2^7 - 1
short   16bit-Ganzzahl    -2^15 .. 2^15 - 1
int     32bit-Ganzzahl    -2^31 .. 2^31 - 1
long    64bit-Ganzzahl    -2^63 .. 2^63 - 1
```

4.1.2. Gleitpunkttypen (nur a/2^n exakt darstellbar!)

```
float         -3.4*10^38 bis +3.4*10^38          Etwa 7 Stellen
double        -1.7*10^308 bis +1.7*10^308        Etwa 16 Stellen
```

4.1.3. Logisch

```
boolean       true oder false (Sind Konstanten, keine Schlüsselwörter)
```

4.1.4. Zweierkomplement

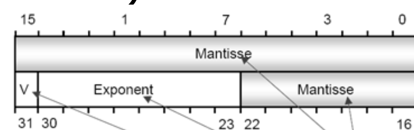
- Erstes Bit gibt das Vorzeichen an

```
Umrechnung von Binär -> Dez

1)     10100111                      > Zahl ist negativ
2)     01011000                      > Alle Bits invertieren
3)     01011001                      > Addiere die Zahl 1 (da negativ)
4)     2^0 + 2^3 + 2^4 + 2^6 = 89    > Summe bilden
5)     -89                           > Minuszeichen anhängen

Umrechnung Dez -> Binär
1)     -27                           > Minuszeichen entfernen
2)     00011011                      > Zahl in Binärsystem umrechnen
3)     00011010                      > Subtrahiere die Zahl 1 (da negativ)
4)     11100101                      > Alle Bits invertieren
```

4.1.5. Gleitpunkttypen (umrechnen)



float: 1 Bit Vorzeichen V, 8 Exp., 23 Mantisse (4 Bytes)
double: 1 Bit Vorzeichen V, 11 Exp., 52 Mantisse (8 Bytes)

```
Interne Darstellung von 3.14f

Vorzeichen-Bit: 0

Exponent (Normiert auf [1.0 bis 2))
3.14 * 2^0
1.57 * 2^1   => 1 + 127 =      0000 0001
                               0111 1111
                               ---------
                               1000 0000
Mantisse
0.57   //0.57*16   > Ausgangszustand, 1.57 – 1 = 0.57
9.12   //0.12*16   > 9   = 1001
1.92   //0.92*16   > 1   = 0001
14.72  //0.72*2    > 14  = 1110
..     ..          ..    ..

Total
0|100 0000 0|100 1000 1111 0…
4     0    4     8    F    ?   => Achtung, Little Endian: F? 48 40
```

4.2. Referenztypen

- Referenzen („Fernsteuerung"), Array, Schnittstelle, Enum

4.2.1. Referenzen

```
/**
* Objekt wird auf dem Heap erstellt
* Referenzvariable auf dem Stack
*/
Punkt p1,p2,p3;
p1 = new Punkt(1);
p3 = new Punkt(3); //new-Operator gibt Referenz auf das Erzeugte Objekt
p2 = p1; //Kopiert Referenz, nicht das Objekt
p1 = null; //Zugriff auf ursprüngliches p1-Objekt geht verloren!
```

4.2.2. Array

- Arrayvariable = Referenz!

```
int arrValues[] = new int[5];
arrValues[0] = 12;

double arrValues2[] = {2.0,2*2,5.0,0.0};
arrValues2.length; //4

int arrValues3[][] = new int [5][5];
arrValues3[0][0] = 1;
```

4.2.3. Enum

- Aufzählungstyp

```
public class Ampel {
//Müssen in einer Klasse eingebettet sein
public enum AmpelFarbe {ROT, GELB, GRUEN};
}
AmpelFarbe farbe;
farbe = AmpelFarbe.ROT;
```

4.3. Variabeltypen

4.3.1. Klassenvariable (static)

- Eine Klassenvariable gibt es in einer Klasse nur einmal
- Lebt vom Laden der Klasse an, bis diese nicht mehr gebraucht wird
- Gehören zur Klasse und befinden sich in der Method Area (ausserdem: Code der Methoden)
- Zugriff erfolgt üblicherweise über den Klassennamen (clsBeispiel.ichBinStatic)
  - Könnte auch über Objekte der Klasse zugegriffen werden, das ist aber kein guter Prog-Stil

4.3.2. Instanzvariable

- Für jedes Objekt angelegt, Normalfall
- Lebt solange wie das Objekt

4.3.3. Lokale Variablen

- Leben nur solange, wie der sie umfassende Block ({…})
- Stack

4.3.4. Konstanten (final)

- Konstante kann genau einmal einen Wert erhalten
- Werden für gewöhnlich in GROSSBUCHSTABEN geschrieben

4.4. String-Klassen

4.4.1. String

```
String strLeer = new String();
String strName = new String(„Anja");
String strName2 = „Anja";
int intLen = strName.length(); //4
```

```
if (strName.equals(strName2)) {
 // bei strName == strName2 werden Referenzen verglichen!
}
```

4.4.2. StringBuffer / StringBuilder

- Genau dieselben Funktionen
- Ausser: StringBuffer ist ThreadSafe, StringBuilder ist schneller
- Achtung: Zum Vergleich müssen die StringBuffer in String umgewandelt werden!
- Hinweis: Viel Schneller als String, wenn die Texte oft geändert werden!

```
StringBuffer strTextA = new StringBuffer(„Bio“); //ok
//StringBuffer strTextB = „Bio“; // wird nicht funktionieren!
strTextA.append(„logisch“); //Text anhängen

if (strTextA.toString().equals(strTextB.toString())) {
//Vergleichen mit STRING
}
```

4.5. Wrapper-Klassen

Wrapperklassen sind Klassen, die dazu dienen, eine Anweisungsfolge oder einen einfachen (elementaren) Datentyp in die Gestalt einer Klasse zu bringen.

- Beispiel: int => Integer
- Beispiel: short => Short

4.6. Boxing

- Boxing: Erstellen eines Wrapper-Objekts aus einem einfachen Datentyp
  - Integer wi = new Integer(i); //manuell
  - Integer wi = i; //automatisch, seit JDK 5.0
- Unboxing: Extrahieren eines einfachen Datentyps aus einem Wrapper-Objekt
  - int i = wi.intValue(); //manuell
  - int i = wi; //automatisch, seit JDK 5.0

4.7. Zugriffsmodifikatoren

	Datenfeld	**Methode**	**Konstruktor**	**Klasse**	**Schnittstelle**
abstract		x		x	x
final	x	x		x	
native		x			
private	x	x	x		
protected	x	x	x		
public	x	x	x	x	x
static	x	x		x	x
synchronized		x			
transient	x				
volatile	x				

5. Ausdrücke und Operatoren (200)

5.1. Operatoren

- Siehe Anhang

5.2. Ausdruck / Anweisung

- Ausdrücke haben in Java stets einen Rückgabewert. Alles das, was einen Wert zurückliefert, stellt einen Ausdruck dar. Anweisungen (try, catch, throw, …) haben keinen Rückgabewert.

5.3. Nebeneffekte

5.3.1. Nebeneffekt von Operatoren

```
int u=1, v;
v = u++; //Nebeneffekt: u ist 2.
```

5.3.2. Nebeneffekt von Methoden

```
p.setX(12); //Nebeneffekt: x-Wert von p wird 12.
```

5.4. Auswertungsreihenfolge

- Teilausdrücke werden strikt von links nach rechts ausgewertet
- Teilausdrücke in Klammern werden zuerst ausgewertet
- Dann werden Ausdrücke mit unären Operatoren ausgewertet
  - Unäre Operatoren immer von rechts nach links ausgewertet

```
byte x=5;
System.out.println(-~x); //gibt 6
/**
*       0000 0101
*       1111 1010 (~ => invertieren, siehe Bitoperatoren)
*       0000 0110 (- => negieren und +1, siehe auch „Zweierkomplement“)
*/
```

- Schliesslich werden Teilausdrücke mit mehrstelligen Operatoren ausgewertet

5.4.1. Assoziativität

- Sagt, wie Operatoren gleicher Priorität abgearbeitet werden.

```
a – b + c          =>    (a – b) + c
a = b = c = 3      =>    a = (b = (c = 3))
```

5.4.2. Fieser Programmierer

```
a = 10;
b = a++ + ++a; //22 -> 10 + 12
```

5.5. Sonderfälle

```
1   / 0      -> Ganzzahlen: Ergibt Exception (Division by Zero)
1.0 / 0.0    -> Gleitpunktzahlen: Ergibt Infinity (Unendlich)

1   % 0      -> Ganzzahlen: Ergibt Exception (Division by Zero)
1.0 % 0.0    -> Gleitpunktzahlen: Ergibt NaN (Not a Number)

0   / 0      -> Ganzzahlen: Ergibt Exception (Division by Zero)
0.0 / 0.0    -> Gleitpunktzahlen: Ergibt NaN (Not a Number)

-7 %  3 = -1 -> Resultat hat das Vorzeichen des Divisors (-7)
 7 % -3 =  1 -> Resultat hat das Vorzeichen des Divisors (+7)
```

5.6. Bitoperatoren

```
 9     1001         9     1001         9     1001         9     1001
&3     0011        |3     0011        ^3     0011        ~9     0110
=1     0001       =11     1011       =10     1010
```

5.7. Bit-Shift-Operatoren

5.7.1. Linksshift-Operator

- A << B: Es werden B Bitstellen von A nach links geschoben. Die höchstwertigen Bits gehen verloren.

```
byte a;
a = 5;        //0000 0101 = 5
a = a << 2;   //0001 0100 = 20
```

5.7.2. Vorzeichenbehafteter Rechtsshift-Operator

- A >> B: Es werden B Bitstellen von A nach rechts geschoben. Niederwertigste Bits von A gehen verloren. Bei positiven Zahlen werden 0 nachgeschoben, bei negativen 1!

```
byte a;
a = 5;        //0000 0101 = 5
a = a >> 2;   //0000 0001 = 1
a = -5;       //1111 1011 = -5 (siehe Zweierkomplement)
a = a >> 2;   //1111 1110 = -2 (siehe Zweierkomplement)
```

5.7.3. Vorzeichenloser Rechtsshift-Operator

- A >>> B: Es werden B Bitstellen von A nach rechts geschoben. Niederwertigste Bits von A gehen verloren. Es werden dabei stets 0-Bits von links nachgeschoben!

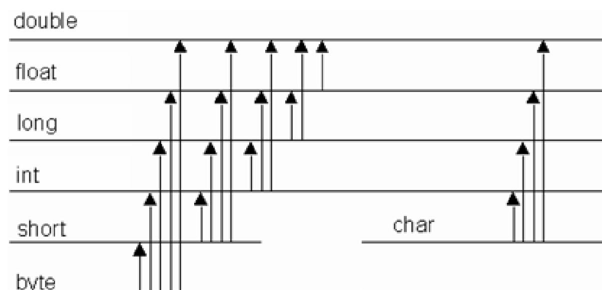
```
byte a;
a = -5;       //1111 1011 = -5
a = a >>> 2;  //0011 1110 = 62
```

5.8. Casten

5.8.1. Implizites Casten

- Wird vom Compiler automatisch gemacht. (kleinerer Typ in grösseren Typ)

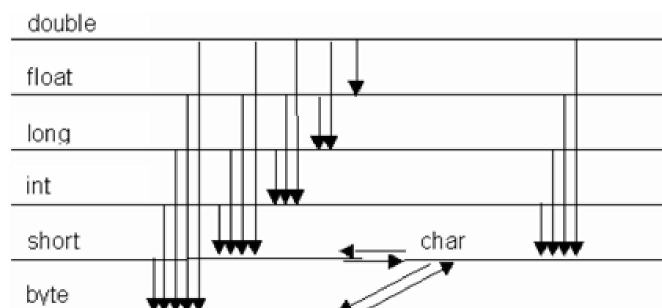
```
char charBlubb = 10;
int intBlubb = charBlubb;
```



5.8.2. Explizites Casten

- Müssen vom Programmierer erzwungen werden
- Double -> Float = Genauigkeitsverlust oder nicht darstellbar
- boolean Werte können nicht konvertiert werden

```
double dblBlubb = 2.14;
int intBlubb = (int) dblBlubb; //2
```



6. Kontrollstrukturen

6.1. Block

- Ein Block ist eine Sequenz von Anweisungen, die mit geschweiften Klammern zusammengehalten werden

```
{ //Ich bin ein Block }
```

6.1.1. Verschachtelte Blöcke

```
{
 int intBeispiel1 = 1;
 {
 //Innerer Block, intBeispiel 1 ist sichtbar
       int intBeispiel2 = 2;
 }
 //intBeispiel2 ist hier nicht mehr sichtbar!
}
```

6.2. If..Then..Else

```
if (intCount == 2) {
 // true
} else {
 // false
}

if (intCount == 2) {
 //intCount ist 2
} else if(intCount == 1) {
 //intCount ist 1
} else {
 //Weder noch
}
```

6.3. Switch

```
switch(intCount) {
 case 1:
 //...
 break; //Wichtig, ohne Break geht’s im switch weiter!
 case 2:
 //...
 break;
 default: //Optional, Muss nicht am Ende stehen!
}
```

6.4. Schleifen / Iterationen

6.4.1. While

```
while (intCount < 2) {
 //Mach was
}
```

6.4.2. Do.. While

- Wird mindestens einmal ausgeführt

```
do {

} while(intCount < 2)
```

6.4.3. For

```
for (int i=0;i<5;i++) {
 //i (Laufvariable) ist nur im Block gültig!
}
```

6.4.4. Endlosschleifen

```
while (true) {}

for (;;) {}
```

6.4.5. For each

```
for (String strValue : args) {
 System.out.println(strValue); // args Muss ein String-Array sein
}
```

6.5. Break

- Mit break wird eine Schleife oder eine Switch-Anweisung abgebrochen.
- Es wird unmittelbar hinter dem Ende der Schleife / Switch fortgefahren

```
while (true) {
 break;
}
//Nach break wird hier fortgefahren
```

6.6. Continue

- Mit continue wird direkt ans Ende der Schleife gesprungen und die nächste Iteration gestartet

```
for (int i=0;i<5;i++) {
 //tu was
 if (i==3) {
  continue;
 } else {
  //...
 }
 //Continue springt zum Schleifenende
}
```

6.7. Sprungmarken

```
{
 Marke: //Sprungmarke

 break Marke; //Spring zu Marke hoch
 //continue Marke; -> Funktioniert auch
}
```

7. Blöcke und Methoden

7.1. Gültigkeit / Sichtbarkeit

- Gültig heisst, dass an einer Stelle im Programm der Name einer Variable bekannt ist
- In Java sind identische Namen in inneren und äusseren Blöcken nicht zugelassen!
- Ein Datenfeld kann gültig, aber nicht sichtbar sein

7.1.1. Beispiel: Sichtbarkeit

```
public class Punkt {
 privat int x;

 public Punkt(int x) {
       this.x = x; //Parameter x verdeckt Datenfeld x -> Sichtbarkeit
 }
}
```

7.2. Methoden

- Eine Methode soll eine genau definierte Teil-Aufgabe lösen
- Return-Anweisung beendet die Methode. Diese kann weggelassen werden, falls Rückgabe-Typ void.
- Methodenkopf definiert formale Parameter
    - Compiler weist beim Aufruf aktuelle Parameter zu
- Es wird immer der Wert kopiert (call by value)
    - Wird eine Referenz übergeben, kann das Objekt auch von innerhalb der Methode manipuliert werden. Es wird aber nur die Referenz kopiert, das Objekt NICHT!

```
[MODIFIKATOREN] Rückgabewert MethodenName([Parameterliste]) {}

private int getDrugs(int intAmount) {
 return intAmount;
}
```

7.2.1. Rückgabewerte

- Es kann genau ein einziger Rückgabewert mittels return zurückgegeben werden (oder keiner)

7.2.2. Finale Methoden

- Finale Methoden lassen sich nicht überschreiben (Bei Vererbung)
- Schlüsselwort: final

7.3. Polymorphie

- Dieselbe Operation mit denselben Parametern kann in verschieden Klassen implementiert werden.

```
Klasse Integer     =>    Methode toString()
Klasse StrinBuffer =>    Methode toString()
Klasse Person      =>    Methode toString()
```

7.4. Überladen von Methoden

- Verschiedene Methoden mit gleichem Methodennamen, aber verschiedenen Parameterlisten innerhalb einer Klasse
- Es ist nicht möglich, in der gleichen Klasse zwei Methoden mit gleichen Namen und gleicher Parameterliste, aber unterschiedlichen Rückgabetypen zu erstellen. Grund: Der Rückgabewert muss nicht abgeholt werden.

7.5. Variabel lange Parameterliste

- Variabler Teil nur als letztes Element der Parameterliste
- Zugriff auf Liste wie auf Array

```
public void printExample(int... arrValues) {
 System.out.printf(arrValues.length);
//Zugriff auf Elemente mit arrValues[i];
}

//Aufruf
refMuster.printExample(1,2,3,4,5); //Gibt 5 Zurück
```

8. Klassen und Objekte

8.1. Allgemeines

8.1.1. Definition einer Klasse

- Namen der Klasse
- objekt- und klassenbezogene Datenfelder
- objekt- und klassenbezogene Methoden
- Sichtbarkeit der Datenfelder und Methoden

8.1.2. Instantierung

- Erzeugung von Objekten
- Objekte sind Variablen, die sich gemäss den in der Klasse beschriebenen Angaben verhalten

8.1.3. Information Hiding

- Um die Konsistenz und Sicherheit der Objekte zu gewährleisten, sollte eine Klasse keinen direkten Zugriff auf die Datenfelder des Objektes erlauben, sondern nur über Klassenmethoden (private / protected)

8.1.4. Identifikation einer Klasse

- Funktioniert mittels des instanceof-Operators

```
if (refA instanceof Blubb) { }  //True
if (refA instanceof Blobb) { } //True, wenn Blobb von Blubb abgeleitet!
```

8.2. This

- Der Programmierer will aufmerksam machen
- Zugriff auf verdecktes Datenfeld
- Aktuelles Objekt (Referenz darauf) als Rückgabewert
    - Verketten von Methodenaufrufe für dasselbe Objekt

```
clsBeispiel getObject() {
 return this;
}

//Aufruf b1.getObject().getValues();
```

8.3. Initialisierung

- Klassenvariablen und Instanzvariablen werden automatisch mit Default-Werten initialisiert
- Lokale variablen werden NICHT initialisiert -> guter Stil: Immer ALLES manuell initialisieren

8.3.1. Defaultwerte für Datentypen

boolean	false	char	\0000
byte	0	short	0
int	0	long	0
float	0.0f	double	0.0d
Referenz	null		

8.3.2. Initialisierungsblock

Ein Initialisierungsblock wird nur einmal ausgeführt, beim Erzeugen, VOR dem Konstruktor

- Klassenvariablen

```
class Init {
 static {
       int a,b,c=3; a=b=7;
 }
}
```

- Instanzvariablen

```
class Punkt3 {
 private x,y {
       x = y = 1;
 }
}
```

8.4. Konstruktoren

- Beim Erzeugen eines Objektes, wird zuerst der Speicherplatz angelegt, anschliessend der Konstruktor der Klasse aufgerufen
- Konstruktor trägt immer den Namen der Klasse
- Es können mehrere Konstruktoren mit verschiedenen Parameterlisten definiert werden
- Ein Konstruktor wird NIE vererbt
  - Aufruf jedoch über super(), muss erste Anweisung sein
  - Ohne super() wird implizit der Default-Konstruktor aufgerufen. Kann Fehler erzeugen!

```
 class Blubb extends Blobb {
       public Blubb () {
             super();
       }
 }
```

- Ein Konstruktor kann andere Konstruktoren in der selben klasse mittels this() aufrufen

```
class Blubb {
 public Blubb(String strName) {

 }
 public Blubb() {
       this(„Kein Name“);
 }
}
```

- Ein Konstruktor hat keinen Rückgabewert
- Sobald nur ein einziger, selbst geschriebener Konstruktor existiert, ist der vom Compiler zur Verfügung gestellte Default-Konstruktor nicht mehr vorhanden

8.5. Verhindern der Instantierung (Singleton)

- In vielen Fällen darf nur eine einzige Instanz einer Klasse existieren

```
class Singleton {
 private static Singleton refInstance;

 private Singleton() {

 }

 public static Singleton getSingleton() {
       if (refInstance == null) {
             refInstance = new Singleton();
       }
       return refInstance;
 }
}
```

8.6. Freigabe von Speicher

- Bei einer Speicherbereinigung werden die nicht mehr referenzierten Objekte aus dem Heap entfernt
  - Beziehungsweise: Ihr Platz wird zum Überschreiben freigegeben

```
refObjekt = null;
```

8.6.1. Garbage Collector

- Alle nicht referenzierten Objekte werden aus dem Heap entfernt

```
System.gc(); //Manueller Aufruf des GC
```

8.6.2. Finalize

- Entfernt der GC ein Objekt aus dem Speicher, so wird zuvor die Methode finalize() für dieses Objekt abgearbeitet
- Wird die finalize()-Methode zuvor manuell aufgerufen, so wird diese vom GC NICHT mehr aufgerufen

8.7. Spezielle Klassen

8.7.1. Object

- Jede Klasse / Array wird von der Klasse Object abgeleitet, ohne explizite Angabe im Programm
- Dadurch erhält jede Klasse automatisch die Methoden der Klasse Object

```
public String toString()
public boolean equals(Object obj)
protected Object clone()
protected void finalize() throws Throwable
```

8.7.2. Class

- Jeder Typ in einem Java-Programm wird in der virtuellen Maschine durch ein Objekt der Klasse Class repräsentiert

```
Class refClass = refObj.getClass();
System.out.println(„Name:“+refClass.getName());
```

- Beispiel zur Erzeugung eines Objektes OHNE new-Operator

```
Class c = Class.forName(„Klassenname“);
Klassenname k = (Klassenname)c.newInstance();
```

8.8. Finale Klassen

- Finale Klassen sind Klassen, von denen sich keine weiteren Klassen mehr ableiten lassen
- Alle Methoden und Attribute sind automatisch auch final

```
final class Blubb {…}
```

8.9. Abstrakte Klassen

- Schlüsselwort abstract
- Eine abstrakte Basisklasse kann nicht instantiert werden
- Ist auch nur eine einzige Methode abstrakt, dann muss auch die ganze Klasse abstrakt sein!

8.10.Generische Klassen

- Methoden und Schnittstellen beinhalten Typparameter als Platzhalter für konkrete Datentypen

```
public class Beispiel<T> {
 private T x;
 private T y;
 public Beispiel(T x, T y) {
  this.x = x;
  this.y = y;
 }
}
```

9. Vererbung und Polymorphie

9.1. Vererbung

9.1.1. Definition

- Die abgeleitete Klasse (Kindklasse) erbt alle Eigenschaften (Methoden und Attribute) der Basisklasse (auch Elternklasse genannt)
- Die Vererbung ist eine „is a"-Beziehung
- Eine Vererbung stellt eine Spezialisierung dar
- Mit dem Konzept der Vererbung können Wiederholungen im Entwurf vermieden werden

9.1.2. Verdeckte Datenfelder

- Verdecken heisst, wenn in der Sohnklasseein Attribut angelegt ist, dass denselben Namen trägt wie ein von der Vaterklasse geerbtes Attribut

```
//Eigenes Datenfeld
x;
this.x;
//Geerbtes Datenfeld (verdeckt);
super.x;
((Person)this).x;
//Achtung: super.super gibt es nicht, da hilft nur Casting
```

9.1.3. Überschreiben von Methoden

- Überschreiben heisst, wenn in der Sohnklasse eine Methode mit gleichem Namen angelegt wird
  - identische Signatur & Rückgabewert
- Überschrieben wird zur Verfeinerung oder Optimierung
- Es kann auch über super.MethodenName() auf die Methode der Elternklasse zugegriffen werden
  - Spart Tipparbeit!

9.1.4. Beispiel

```
class Person {
 private int intAlter;
 private String strName;

 protected String getName() {…};
 protected int getAlter() {…};
}

class Student extends Person {
 private int intMartikelNummer;
 protected int getMartikelnummer() {…};
}

//Aufruf
Student refKaelin = new Student();
refKaelin.getName(); //Kindklasse hat Zugriff auf Funkt. der Elternklasse
```

9.1.5. Konsistenzhaltung

- Bytecode (.class) und Quellcode (.java) müssen konsisten gehalten werden
- Wenn eine Basisklasse geändert wird, müssen alle Kindklassen neu übersetzt werden

9.2. Polymorphie

9.2.1. Definition

- Polymorphie von Methoden
  - Ein und derselbe Methodenaufruf in verschiedenen Klassen kann klassenspezifisch sein (toString())
- Polymorphie von Objekten
  - Ein Kindobjekt ist auch vom Typ jeder zugehörigen Oberklasse
  - Anstelle eines Objektes einer bestimmten Klasse kann stets auch ein Objekt einer abgeleiteten Klasse stehen (Liskov Substituion Principle)

9.2.2. Object

- Jedes Objekt ist von der Klasse Object abgeleitet

9.2.3. Beispiel

```
//Erweiterung des Vererbungs-Beispiels
public void printPerson(Person refPerson) {…};
public void printAlter(Object refPerson) {…}; //Achtung, Typensicherheit!

//Aufruf
Student refKaelin = new Student();
printPerson(refKaelin); //Ein Student-Objekt wird übergeben, kein Fehler!
```

9.2.4. Typenkonvertierung

- Up-Cast

```
Person refPerson = new Student(); //Kein Problem
```

- Down-Cast

```
Student refStudent = (Student) new Person(); //Cast nötig
```

10. Pakete

10.1. Das Wichtigste

- Logisch zusammengehörende Klassen werden in einem Paket zusammengefasst
- Gröbste Strukturierungseinheit in der OO-Welt
- Jedes Paket definiert einen eigenen Namensraum
  - In verschiedenen Paketen können Klassen / etc. mit demselben Namen existieren
- Pakete haben eine eigene Sichtbarkeit
  - Ohne Angabe der Sichtbarkeit (default) ist eine Klasse / etc. aus allen Dateien desselben Pakets sichtbar
- Pakete können andere Pakete enthalten
- Paketstruktur entspricht meistens 1:1 der Dateistruktur
- Pakete bilden eigenen Bereich für Zugriffsschutz
- Nachteil: Angabe des Classpaths

10.2. Deklaration

- Paketnamen werden klein geschrieben
- Pakete für einen grösseren Benutzerkreis müssen eindeutig sein
  - Lösung: Paketnamen bilden aus eigenem Internet-Domain, und zwar in umgekehrter Reihenfolge: ch.hsr.i.prog1.kaelin
- Pro Datei ist nur eine Paket-Deklaration möglich
- Package-Deklaration muss die erste Anweisung in einer Java-Datei sein

```
package <name>.<subname>;
```

10.3. Import von Paketen

- Ermöglicht Schreibweise von Klassen ohne qualifizierten Klassen-Namen
- Import steht nach der Paket-Deklaration
- Mit „.*“ werde keine Unterpakete importiert!
- Java.lang wird automatisch importiert

```
package beispiel2; //importiert Package „beispiel2“, enthält klasse1
import beispiel1.*; //importiert alle public Klassen / Schnittstellen

public class clsBeispiel {
 public static void main(String args[]) {
       //beispiel2.klasse1 refKlasse1 = new beispiel2.klasse1;
       klasse1 refKlasse1 = new klasse1;
 }
}
```

11. Exception Handling

11.1. Grundlagen

- Eine Exception (Ausnahme) ist ein Ereignis, welches während der Ausführung eines Programms auftritt. Dieses Ereignis unterbricht den normalen Instruktionsfluss.

11.1.1. Checked Exception

- Im Falle einer zu berücksichtigenden Ausnahme erzwingt Java eine Fehlerbehandlung. Wird diese Unterlassen, so wird diese beim Kompilieren gemeldet.
  - Alles was von Exception abgeleitet ist, aber nicht in RuntimeException liegt

11.1.2. Unchecked Exception

- Können behandelt werden, müssen aber nicht.
  - RuntimeException (NullPointerException, IndexOutOfBoundsException, ArrayOutOf..)
  - Error: Ein Programm sollte i.d.R. nicht versuchen, einen solchen Fehler aufzufangen. Schwerwiegender Fehler! (OutOfMemory)

11.2. Try/Catch

- Mit throw können alle Klassen „geworfen" werden, die von Throwable erben

```
try {
 //mach was böses!
 ExceptionName refEx = new ExceptionName();
 throw refEx; //Wirft eine Exception vom Typ ExceptionName
} catch (ExceptionName exInstanz) {
 //Fehler wurde aufgefangen, behandle diesen
 System.exit();
} finally {
 //Ist optional, falls mindestens ein Catch vorhanden ist
}
```

```
//Wir erstellen eine eigene Exception
class ExceptionName extends Exception {
 ExceptionName() {
       super(„Fehlertext.“); //Fehlermeldung an Konstruktor von Exception
 }
}
```

11.3. Propagieren von Exceptions

- Eine Methode muss Exceptions, welche sie auslöst, nicht selber abfangen. Dies kann auch in einer sie aufrufenden Methode erfolgen.

```
public int getValues() throws Exception {
 //propagiere Exception, keine Behandlung innerhalb der Methode
}

//Aufruf
try {
 intNumber = getValues(); //Wirft Exception, muss behandelt werden!
}
```

11.4. Vorteile

- Saubere Trennung des Codes in „normalen" Code und in Fehlerbehandlungscode.
- Der Compiler prüft, ob Checked Exceptions vom Programmierer abgefangen werden. Nachlässigkeiten werden bereits zur Kompilierzeit und nicht erst zur Laufzeit entdeckt.
- Das Propagieren einer Exception erlaubt, diese auch in einem umfassenden Block oder einer aufrufenden Methode zu behandeln.
- Da Exception-Klasse in einem Klassenbaum angeordnet sind, können je nach Bedarf spezialisierte Handler oder generalisierte Handler geschrieben werden.

12. Schnittstellen

- Sprachmittel zur Trennung von Entwurf, Spezifikation und Implementierung
  - Spezifierende Sicht: Schnittstelle einer oder mehrerer Klassen, das WAS?
  - Implementierende Sicht: Aufbau der Methoden, das WIE?
- Eine Schnittstelle ist immer implizit public abstract
  - Private nicht möglich, da unsinnig
- Eine Schnittstelle kann nur Methodendeklarationen und Konstanten enthalten
- Im Gegensatz zu Klassen gibt es Mehrfach-Vererbung

12.1. Interface / Implements

- Eine Klasse muss immer alle Methoden einer Schnittstelle implementieren!
- Eine Klasse kann mehrere Schnittstellen implementieren

```
interface iBeispiel { //Was?
 public int getX();
 public void setX(int intX);
}

class Beispiel implements iBeispiel { //Wie?
 public int getX() {…
 }

 public void setX(int intX) {…
 }
}
```

12.2. Vererbung von Schnitstellen

```
interface iBeispiel2 extends iBeispiel1 {
 public int getY();
}
```

12.3. Probleme mit Schnittstellen

- Folgende Probleme können beim Implementieren von mehreren Schnittstellen auftreten

12.3.1. Gleichnamige Konstanten

- Qualifizierte Namen verwenden

12.3.2. Exakt gleiche Methoden

- Die Methode wird nur einmal implementiert

12.3.3. Gleiche Methoden mit unterschiedlichen Exceptions

- Die eine der beiden unterschiedlichen Exceptions muss von der anderen abgeleitet sein
- Die implementierte Methode muss dann die spezialisierte Exception werfen

12.3.4. Unterschiedliche Rückgabewerte bei Methoden

- Nicht möglich, da Methoden nicht anhand von Rückgabewerten unterschieden werden können.

12.4. Unterschiede: Abstrakte Klasse / Schnittstelle

- Abstrakte Basisklasse
  - Variablen, Konstanten, sowie implementierte und abstrakte Methoden enthalten
  - nur eine Basisklasse beerben
- Schnittstelle
  - nur Konstanten und abstrakte Methoden enthalten
  - mehrfach vererbbar

12.5. Spezielle Interfaces

12.5.1. Comparable

- Gleichheit ist wichtig für Tests und Sortierverfahren. Aber was ist Gleichheit bei Objekten?
- Ist das Objekt A, dessen compareTo()-Methode mit Objekt B als Parameter aufgerufen wird..
  - kleiner => -1
  - gleich => 0

   - grösser => 1

```
public interface Comparable {
 public int compareTo (Object o);
}

//Nachfolgend ein Beispiel
public class Blubb implements Comparable {
 int x;

 public int compareTo(Object o) {
       if (this.x < o.x) {
             return -1;
       } else if (this.x > o.x) {
             return 1;
       } else {
             return 0;
       }
 }
}
```

12.5.2. Cloneable

- Erstellt eine exakte Kopie eines Objektes
   - Alle einfachen und benutzerdefinierten Datenfelder werden kopiert
- Überprüfung, ob Objekt kopierbar:

```
if (<MyClass> instanceof Cloneable) {…}
```

- Flache Kopie: Einzig die Attributwerte des Objektes werden kopiert. Die referenzierten Objekte werden nicht kopiert, sondern nur die Referenzen auf die Objekte.
- Tiefe Kopie: Neben den Attributwerten des Objekts werden auch alle referenzierten Objekte kopiert

12.5.3. Serializable

- Werden Werte der Datenfelder eines Objektesin einen Bytestrom überführt, der sich wieder rekonstruieren lässt, so spricht man von Objektserialisierung.
- Für die Überführung der Datenfelder eines Objektes in einen Bytestrom ist die Klasse ObjectOutputStream zuständig. Für das Rekonstruieren die Klasse ObjectInputStream.
- Die Schnittstelle Serializable kennzeichnet ein Objekt als serialisierbar, implementiert aber selber keine Methoden!
- Datenfelder mit dem Schlüsselwort transient werden nicht serialisiert!
- Hat man in einem verteilten System eine Klasse, welche Serializable implementiert, erst einmal verbreite, ist man an die ursprüngliche Implementierung der Klasse gebunden. Jede Änderung oder Erweiterung führt dazu, dass mit der Klasse keine Objekte eingelesen werden können, die mit einer Vorgängerversion geschrieben wurden. Umgehen kann man die Problematik, in dem man die serialVersionUID für eine Klasse manuell festlegt!

```
public class Beispiel implements Serializable {
 private static final long serialVersionUID = 123456;
 private int intBlubb;
 transient int intBlobb; //Ich werde nicht serialisiert
}

public class Serial {
 public static void main (String args[]) {
 ObjectOutputStream out = new ObjectOutputStream(
                                 new FileOutputStream(„test.ser“));
 Beispiel refBeispiel = new Beispiel();
 out.writeObject(refBeispiel); //serialisiert Objekt
 out.close();

 ObjectInputStream in = new ObjectInputStream(
                                 new FileInputStream(„test.ser“);
 refBeispiel = (Beispiel) in.readObject(); //lesen des Objekts
 }
}
```

13. Ein-/Ausgabe und Streams

13.1. Klassifizierung

	Input	Output
Byte	InputStream	OutputStream
Character	Reader	Writer

- Objekte von Bytestream-Klassen lesen und schreiben Bytes, also Werte vom Typ byte. Objekte dieser Klassen können selbst auch Daten vom Typ char und String – also Zeichen – verarbeiten, jedoch geht dabei das jeweils höherwertige Byte eines jeden Zeichens verloren. Deshalb muss darauf gachtet werden, dass für die korrekte Ein- und Ausgabe von Zeichen, die für ihre Darstellung mehr als 1 Byte benötigen, Characterstream-Klassen eingesetzt werden.
- Objekte von Characterstream-Klassen schreiben und lesen Characters, also char. Für jedes Zeichen vom Typ char werden zwei Byte benötigt.

13.1.1. Sink-, Spring- und Processingstreams

- Ein Objekt einer Sinkstream-Klasse kann Daten in eine Senke schreiben
- Ein Objekt einer Springstream-Klasse kann Daten direkt aus einer Datenquelle lesen
- Ein Objekt einer Processingstream-Klasse benutzt intern ein Objekt einer Sinkstream- oder Springstream-Klasse und erweitert deren Funktionalität. Jede Processingstream-Klasse ist von einer der Klassen abgeleitet und aggregiert gleichzeitig ein Objekt des folgenden Typs:
  - Byte-Inputstream: privates Feld vom Typ InputStream
  - Byte-Outputstream: privates Feld vom Typ OutputStream
  - Character-Inputstream: privates Feld vom Typ Reader
  - Character-Outputstream: privates Feld vom Typ Writer

13.1.2. Read-Funktion: int-Wert als Rückgabe?

- Warum gibt die read()-Funktion einen int-Wert zurück, obwohl byte oder char gelesen werden?
  - Beim Ende des Streams wird „-1" gelesen. Im Typ char könnte das jedoch nicht dargestellt werden.

13.2. Codebeispiele

13.2.1. FileReader / FileWriter

- Nachfolgender Code kopiert eine Datei. Code ist nur schematisch, kompiliert so sicher nicht! (Exceptions, main(), ..)

```
import java.io.FileReader;
import java.io.FileWriter;

FileReader refReader = new FileReader(strFile); //Springstream-Klase
FileWriter refWriter = new FileWriter(strFile); //Sinkstream-Klasse

while ((intZeichen = refReader.read()) != -1) {
 refWriter.append((char)intZeichen);
}

refReader.close();
refWriter.close();
```

13.2.2. ObjectInputStream

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;

//ObjectInputStream = Processingstream-Klasse (aggregiert Objekt)
ObjectInputStream rIn = new ObjectInputStream(new FileInputStream(sPath));
```

14. Collections

Collections sind Zusammenstellungen von Daten, genauer gesagt von Objekten.

14.1. Überblick

		Organisation	Freiheit beim Zugriff	Mechanismus beim Zugriff	Umgang mit Duplikaten	Besondere Eigenschaften
Listen	**ArrayList**	geordnet	wahlfrei	über Index	ja	schneller Zugriff
	LinkedList	geordnet	wahlfrei	über Index	ja	schnelles Einfügen
	Vector	geordnet	wahlfrei	über Index	ja	synchronisiert
	Stack	geordnet	sequenziell	letztes Element	ja	LIFO
Queues	**LinkedList**	geordnet	sequenziell	nächstes Element	ja	
	PriorityQueue	sortiert	sequenziell	nächstes Element	ja	-
Sets	**HashSet**	ungeordnet	wahlfrei	einfacher Test	nein	schneller Zugriff
	TreeSet	sortiert	wahlfrei	einfacher Test	nein	-
Maps	**HashMap**	ungeordnet	wahlfrei	über Schlüssel	Schlüssel nein, Werte ja	schneller Zugriff
	TreeMap	sortiert	wahlfrei	über Schlüssel	Schlüssel nein, Werte ja	-
	LinkedHashMap	geordnet	wahlfrei	über Schlüssel	Schlüssel nein, Werte ja	-

- Geordnet: Elemente werden in derselben Reihenfolge abgespeichert, in der sie eingefügt wurden
- Sortiert: Inhalt nach einer Ordnungsrelation gespeichert (aufsteigend, ..)
- Sequenziell: Es kann nur auf bestimmte Elemente zugegriffen werden
- Wahlfrei: Es kann auf jedes Element der Collection zugegriffen werden

14.1.1. Listen

- Listen sind geordnete Collections, auf die wie auf Arrays an beliebiger Stelle über numerische Indizes zugegriffen werden kann. Im Unterschied zu Arrays können Elemente an beliebiger Stelle eingefügt werden.

14.1.2. Queue

- Eine Liste, die nach dem FIFO-Prinzip abgearbeitet wird

14.1.3. Menge / Set

- Eine ungeordnete Menge von Objekten, wobei jedes nur einmal vorkommen darf

14.1.4. Assoziative Arrays / Maps

- Es wird über einen eindeutigen Schlüssel eines beliebigen Typs zugegriffen, anstatt über einen numerischen Schlüssel. In einer Map werden Schlüssel-Wert Paare repräsentiert.

14.2. Iterator

- Ein Iterator stellt eine standardisierte Methode einer Collection dar, die es erlaubt, auf eine Collection zuzugreifen, ohne den speziellen Typ der Collection zu kennen. Der Iterator erlaubt es, über alle Elemente der Collection zu laufen, sowie festzustellen, ob noch ein weiteres Element in der Collection ist.
- Wurde früher auch Enumeration genannt

14.2.1. Codebeispiel

```
int arrValues[] = {1,2,3,4,5};
List arrList = new ArrayList();

for (int i = 0; i<arrValues.lengh; ++i) {
 arrList.add(new Integer(arrValues[i]));
}

Iterator iter = arrList.iterator();
while (iter.hasNext()) {
 Integer intValue = (Integer) iter.next();
}
```

15. Sonstiges

15.1.printf()

- Leere Zeichen vor dem Punkt werden mit Leerzeichen aufgefüllt
- Der Punkt selber zählt als Zeichen (siehe 2. Beispiel)

```
//3 Stellen, Leerzeichen vorne (10 = 2 Stellen), Dezimal
System.out.printf(„%3d“,10);

//6 Stellen, zwei hinter Komma (xxx.xx), Float
System.out.printf(„%6.2f   %6.2f“,1.0f,2.2f);
```

15.2.Geschachtelte Klassen

- Geschachtelte Klassen werden hauptsächlich bei der Oberflächenprogrammierung benötigt
- Das Konzept der geschachtelten Klassen in Java erlaubt es, in einer Klasse andere Klassen zu definieren.
- Geschachtelte Klassen werden in Java dazu eingesetzt, um Klassen, die für die Implementierung benötigt werden, zu verbergen (Information Hidings).

Innere und äussere Klassen können – in vollkommen symmetrischer Weise – wechselseitig auf ihre Datenfelder und Methoden zugreifen

15.3.Applets

- Applets laufen nicht als eigenständige Programme in einer virtuellen Maschine, sondern sind in eine HTML-Seite eingebettet. Ein in der HTML-Seite enthaltenes Applet wird durch den Browser von einem Webserver geladen und dann von einer im Browser enthaltenen virtuellen Maschine ausgeführt.
- Zum Testes eines Applets ausserhalb eines Browsers kann der beim JDK mitgelieferte Appletviewer verwendet werden
- Alle Applets haben die Klasse java.applet.Applet als Vaterklasse
- Ausführung eines Applets
  - main(): Wird NICHT benötigt!
  - init(): Wird einmalig nach dem Erzeugen eines Applets-Objekts aufgerufen.
  - start(): Nachdem eine HTML-Seite aufgebaut wurde, wird die start()-Methode des Applets ausgeführt. Sie wird auch bei jedem erneuten Anzeigen des Applets aufgerufen.
  - stop(): Nachdem das Applet aus dem sichtbaren Bereich gescrollt, bzw. eine neue HTML-Seite geladen wurde, ruft der Browser die stop()-Funktion auf. Das Applet bleibt geladen, ist jedoch in seiner Ausführung angehalten.
  - destroy(): Diese Methode wird beim Löschen der HTML-Seite aus dem Speicher aufgerufen.
- Andere Methoden
  - paint(): Diese Methode wird von der virtuellen Maschine jedes Mal aufgerufen, wenn ein Neuzeichnen einer Komponente erforderlich ist.

15.4.Swing

- In der ersten Version von Java, war nur eine sehr einfache Klassenbibliothek zur Oberflächenprogrammierung enthalten: AWT (Abstract Window Toolkit).
- Die Weiterentwicklung davon ist Swing. Dies ist eine leichtgewichtige Komponente ohne Betriebssystem-Support (Lightweight). Diese werden von Java selbst gezeichnet, und nicht vom Betriebssystem.
- Mischen von AWT und Swing-Elementen ist nicht zu empfehlen!

15.4.1. Java GUI

- Komponenten: Buttons, Labels, Textboxen..
- Container: Gruppieren enthaltener Oberflächenelemente -> Fenster
- Layout Manager: legen fest, wie die Elemente im Container angesprochen werden können

15.4.2. Panes

- Root-Pane: Unterste Ebene
- Layered-Pane: ContentPane + MenuBar
- ContentPane: Enthält die einzelnen Elemente der Benutzeroberfläche
- Glass-Pane: Normalerweise durchsichtig. Oberste Ebene.

15.4.3. Komponenten

- JFrame: Normales Fenster, mit Buttons zum Schliessen, Minimieren, Maximieren und Systemmenu.
- JDialog: Wie JFrame, aber ohne Minimieren / Maximieren
- JWindow: Blankes Fenster, ohne Buttons, Rahmen oder Titelzeile
- JButton: Schaltfläche
- JCheckBox: Kontrollkästchen, Auswahl für Optionen
- JLabel: Statischer Text auf der grafischen Oberfläche
- JTextField: Textfeld
- JPasswordField: Passwortfeld
- JFormattedTextField: Erweitertes Textfeld, dass die Formatierung des Feldinhaltes vorgibt
- JTextArea: Mehrzeiliges Textfeld
- JList: Darstellung und Auswahl von Objekten aus einer Liste
- JComboBox: Kombination aus Textfeld und Liste
- JTree: Darstellung von Klassen als Bäume
- JTable: Darstellung von Daten in Tabellenform

15.4.4. Container

- JPanel: Container für Komponenten
- JScrollPane: Container für Komponenten mit Scrollbar
- JSplitPane: Container mit zwei Bereichen
- JTabbedPane: Container mit Tabs
- JInternalFrame: Multidokument Container

15.4.5. Menüs

- JMenu: Eigentliches Menu
- JToolTip: Kontextsensitive Hilfe
- JToolbar: Werkzeugkasten

15.4.6. Sonstige Komponenten

- JFileChooser: Dateiauswahl
- JColorChooser: Farbpalette

15.4.7. Layoutmanager

- Mit einem Layoutmanager werden die Komponenten nach bestimmten Vorgaben auf einer Bedienoberfläche, bzw. in einem Container angeordnet

15.4.8. Verarbeitung von Ereignissen

- Unter Ereignisverarbeitung versteht man die asynchrone Verarbeitung von Nachrichten
- Diese Nachrichten werden meist von Benutzern ausgelöst
  - Menu-Auswählen, Minimieren eines Fensters
- Vorteile des Delegations-Prinzips
  - Klare Trennung zwischen Oberflächenelementen und Verarbeitung der Ereignisse
  - Es werden nur die Objekte über Ereignisse informiert, die an der Ereignis-Quelle registriert sind

15.4.9. Delegations-Modell

- Involvierte Elemente
  - Event-Quelle: Ein Objekt, das Events generiert (Button, Menu, …)
  - Event-Objekt: Beschreibt das Ereignis näher und liefert dazu passende Kontextinformationen
  - Event-Empfänger: Ein Objekt, welches auf Events reagieren möchte. Beispielsweise ein Objekt, das über die Betätigung eines Buttons benachrichtig wird.
- Zwei Schritte
  - Empfänger muss sich bei der Eventquelle registrieren, um bei einem eingetretenen Ereignis automatisch benachrichtig zu werden
  - Nachdem ein Event aufgetreten ist, wird es von der Eventquelle an alle registrierten Empfänger weitergegeben (delegiert).

16. Anhang

16.1. Schlüsselwörter (Beispiele)

Kategory	Keyword	Example	Incidents per 100 Lines
Primitive types	boolean	`boolean isOpen = true;`	.82
	byte	`byte i1 = -128;`	.67
	char	`char c = '\uFFFF';`	.27
	short	`short i2 = -32768;`	.14
	int	`int i = -2147483648;`	4.35
	long	`long i8 = -9223372036854775808L;`	.19
	float	`float x4 = -3.402823e+38f;`	.32
	double	`double x = -1.79769313486231e+308;`	.33
Control flow	for	`for (int i=0; i<10; i++) {…}`	.63
	do	`do {…} while (i<10);`	.01
	while	`while (i<10) {…}`	.22
	if	`if (i==10) {…}`	3.41
	else	`else if (i<0) {…} else {…}`	.92
	switch	`switch (i) { case 1: … break; default:… }`	.08
	case		.44
	default		.05
	break	`break label;`	.28
	continue	`continue label;`	.03
	return	`return i;`	2.91
	try	`try { … throw new MyException (); … } catch (MyException ex) {…} finally {…}`	.30
	throw		.38
	catch		.34
	finally		.01
	throws	`void func () throws MyException {…}`	.53

Kategory	Keyword	Example	Incidents per 100 Lines
Modifier	public	`public int i;`	4.65
	protected	`protected int i;`	.64
	private	`private int i;`	1.10
	static	`static int i;`	1.56
	final	`final int i;`	.92
	abstract	`abstract void func ();`	.21
	synchronized	`synchronized (object) {…}`	.26
	native	`native int func ();`	.05
	transient	`public transient int i;`	.07
	volatile	`public volatile int i;`	0
Classes	class	`class A {…}`	.86
	interface	`interface I {…}`	.07
	extends	`class B extends A {…}`	.44
	implements	`class B implements I {…}`	.22
	package	`package de.falkhausen.util;`	.36
	import	`import java.awt.*;`	1.40
Miscellaneous	(true)	`boolean isOpen = true;`	.45
	(false)	`boolean isOpen = false;`	.53
	(null)	`Object obj = null;`	2.00
	void	`void func () {…}`	2.15
	this	`this.x = x;`	1.16
	new	`Object obj = new Object ();`	2.71
	super	`super ("text");`	.43
	instanceof	`if (o instanceof String) String s = (String) o;`	.25

16.2. Operatoren (Beispiele)

Prio-rity	Opera-tor	Name	Asso-ciativity	Example	Result*	Incidents per 100 Lines
1	++	Increment	r	`x++`	3.5 (+ effect)	.77
				`++x`	4.5 (+ effect)	
	--	Decrement	r	`x--`	3.5 (+ effect)	.09
				`--x`	2.5 (+ effect)	
	+	Unary plus	r	`+x`	3.5	?
	-	Unary minus	r	`-x`	-3.5	?
	!	Logical complement	r	`!isOpen`	false	.28
	~	Bitwise complement	r	`~i`	-5	.01
	(type)	Cast	r	`i = (int) x`	3	?
2	*	Multiplication	l	`x * 2`	7.	1.24
	/	Division	l	`x / 2`	1.75	.22
	%	Remainder	l	`x % 2`	1.5	.04
3	+	Binary plus	l	`x + 2`	5.5	(2.70)
				`" " + x + i`	" 3.54"	
	-	Binary minus	l	`x - i`	-0.5	(1.46)
4	<<	Shift left	l	`i << 2`	16	.08
	>>	Shift right	l	`-i >> 2`	-1	.04
	>>>	Shift right ignore sign	l	`-i >>> 2`	1073741823	.02
5	>	greater than	l	`i > x`	true	.36
	<	lesser than	l	`i < x`	false	.86
	>=	greater equal	l	`i >= x`	true	.14
	<=	lesser equal	l	`i <= x`	false	.24
	instanceof	Type check	l	`s instanceof String`	true	.25

*Results are given for the declarations: int i=4, int j=2, double x = 3.5, String s="", boolean isOpen=true.

Prio-rity	Opera-tor	Name	Asso-ciativity	Example	Result*	Incidents per 100 Lines
6	==	Equals	l	`i == j`	false	1.28
				`s == ""`	true	
	!=	Not equal	l	`i != j`	true	1.17
				`s != null`	true	
7	&	Bitwise and	l	`i & j`	0	.18
8	^	Exclusive or	l	`i ^ 5`	1	.01
9	|	Bitwise or	l	`i | j`	6	.10
10	&&	Logical and	l	`isOpen && false`	false	.58
11	||	Logical or	l	`isOpen || false`	true	.33
12	? :	Conditional	r	`i<0 ? -1 : 1`	1	.20
13	=	Assignment	r	`j = i`	4 (+ effect)	9.68
				`o = s;`	"" (+ effect)	
	+=	Plus assignment	r	`j += x`	5 (+ effect)	.27
	-=	Minus assignment	r	`j -= x`	-1 (+ effect)	.09
	*=	Muiltiplication assign.	r	`j *= x`	7 (+ effect)	.02
	/=	Division assign.	r	`j /= x`	0 (+ effect)	0
	&=	Bitwise and assign.	r	`j &= i`	0 (+ effect)	.01
	|=	Bitwise or assign.	r	`j |= i`	6 (+ effect)	.03
	^=	Exclusive or assign.	r	`j ^= i`	6 (+ effect)	0
	%=	Remainder assign.	r	`j %= i`	1 (+ effect)	0
	<<=	Shift left assign.	r	`j <<= i`	32 (+ effect)	0
	>>=	Shift right assign.	r	`j >>= i`	0 (+ effect)	0
	>>>=	Shift right i.s. assign.	r	`j >>>= i`	0 (+ effect)	0

16.3. Sichtbarkeit

Zugriff auf	private	default	protected	public
Klasse A	Ja	Ja	Ja	Ja
Klasse B Gleiches Paket	Nein	Ja	Ja	Ja
Subklasse C Gleiches Paket	Nein	Ja	Ja	Ja
Subklasse E Anderes Paket	Nein	Nein	Ja	Ja
Klasse D anderes Paket	Nein	Nein	Nein	Ja

16.3.1. Sichtbarkeit und Vererbung

- Eine überschreibende Methode (in einer Unterklasse) darf die Zugriffsmodifikatoren nur erweitern, nicht einschränken!