Objektorientierte Konzepte

Information Hiding (Kapselung)

Kapselung: Daten und Methoden sind zusammen in einem Objekt

Information Hiding: Die Aussenwelt soll keine Möglichkeit haben die Daten in einem Objekt direkt zu manipulieren (nur durch definierte Schnittstellen).

Abstraktion (Aggregation)

Konzentration auf das Wesentliche; Aus einem gegeben Problem sollen die Entitäten gefunden werden, zu den Entitäten die nötigen Informationen etc.

"is a"-Hierarchie: Vererbungshierarchie (Generalisierung und Spezialisierung)

"part of"-Hierarchie: Java kennt nur die Aggregation (dabei können die Teile länger leben als das ganze, da sie nicht fest "verschweisst" sind.

Vererbung / Polymorphie

Unterklasse erbt von der Oberklasse (Super-Klasse) alle sichtbaren Methoden und Datenfelder. Java kennt nur die Einfachvererbung.

```
public class Unterklasse extends Oberklasse
```

(Alle Klassen sind von Object abgeleitet)

Polymorphie

- *Polymorphie von Operationen* bedeutet, dass eine Methode in verschiedenen Klassen spezifisch implementiert wird (z.B. toString)

- *Polymorphie von Objekten* gibt es in Vererbungshierarchien: An Stelle einen Objekts kann immer auch ein Objekt einer abgeleiteten Klasse eingesetzt werden, da sich abgeleitete Klassen polymorph verhalten können (also wie die Basisklasse) → **Liskov Substitution Principle**

Singleton Pattern

Objekt exisitiert genau 1 Mal.
Privater Konstuktor + getInstance():

```
public class Uni {
 private static Uni inst = null;
 private Uni() {}

 public static Uni getInstance(){
  if (inst == null) {
   inst = new Uni();
  }
  return inst;
 }
}
```

Aufruf:

```
Uni u = Uni.getInstance();
```

Datentypen/Variablen

Einfache Datentypen

boolean (true, false)
char (16 bit, 0 bis 65535 alle Unicode Zeichen)
byte (8 bit Ganzzahl, -2^7 bis $+2^7-1$)
short (16 bit Ganzzahl, -2^{15} bis $+2^{15}-1$)
int (32 bit Ganzzahl -2^{31} bis $+2^{31}-1$)
long (64 bit Ganzzahl -2^{63} bis $+2^{63}-1$)
float (Gleitpunktzahl, $-3.4*10^{38}$ bis $+3.4*10^{38}$)
double (Gleitpunktzahl, $-1.7*10^{308}$ bis $+1.7*10^{308}$)

Binäre Repräsentation

Zweierkomplement (neg. Zahl)
1. Vorzeichen ignorieren und ins Binärsystem umrechnen
2. Bits invertieren
3. 1 addieren
Von Hand: von rechts her alle 0 und das erste 1 schreiben, dann alles invertieren

Gleitkommazahlen zu Dezimal
Idee: x = Mantisse * $10^{Exponent}$
Neg.Zahl: 1. Bit ist Vorzeichen (1=neg.)
IEEE 754: 1 Bit Vorzeichen + 8/11 Bits Exponent + 23/52 Bits Mantisse → jeweils float/double

Dezimal zu Gleitkommazahlen
-6.5
1 10000001 10100000000000000000000
| | \
| \ Mantisse 23 bit
\ Expoinent 8 bit
Vorzeichen 1 bit

Vorzeichen: bestimmt erstes Bit (1 = neg.)
Exponent:
6.5 durch 2 teilen bis kleiner 2:
6.5 / 2 = 3.25 /2 = 1.625
Exponent = Anzahl Teilungen + 127
=> 2 + 127 = 129 = 10000001

Mantisse:
Eins abzählen (wird immer implizit dazugezählt) und mit 2 multiplizieren bis eine Ganzahl erricht ist:
0.625 * 2 = 1.25 * 2 = 2.5 * 2 = 5

Anz. Schritte bis Ganzahl = verfügbare Bits Ganzahl in den verfügb. Bits dargestellen
5 = 101
anderes Bsp.: nach 5 Schritten 3 -> 00011)

Alles zusammen setzen und mit 0 aufüllen

Klassentypen

Alle Objekte werden durch Referenzvariablen angesteuert. Der new-Operator erstellt dabei ein neues Objekt und weist eine Referenz darauf zurück:

```
Integer zahl = new Integer(3);
```

Arrays

```
int[] numbers = new int[10];
long[] l_arr = {12, 23,74};
```

Index: 0 bis n-1
Länge: array.length (Attribut nicht Methode!)

Initialisierung

In Methoden: Alle lokalen Variablen müssen initialisiert werden, sonst gibt es einen Compiler-Fehler

In Klassen: Nicht unbedingt nötig, falls nicht explizit initialisiert wird vom Compiler initialisiert:

Typ	Wert
boolean	FALSE
char	'\u0000'
byte, short, int, long	0
float	0.0f
double	0.0d
Referenztyp	null

Verdecken

- Eine lokale Variable mit gleichem Namen "verdeckt" die Instanzvariable und muss via **this.variable** erreicht werden.
- Unterklasse definiert ein Datenfeld mit gleichem Namen wie in der Oberklasse. Zugriff auf das Datenfeld der Oberklasse via **super.variable**

String / StringBuffer

Strings sind konstant und können nicht verändert werden. Zwei gleiche Strings werden deshalb vom Compiler nur einmal gespeichert und die **gleiche Referenz vergeben**.

StringBuffer sollten verwendet werden bei häufigen Stringmanipulationen.
String-Repräsentation: toString().
Vergleich: equals()
Verkettung: append()
(bei String mit "+" Operator)

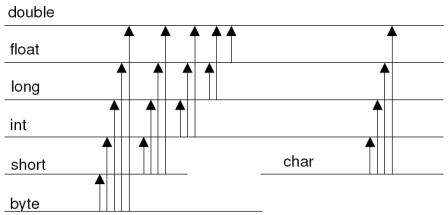
(Auto-) Boxing

Bezeichnet das ein- und auspacken von einfachen Datentypen in Wrapper-Klassen.
Auto-Boxing heisst der Compiler übernimmt das Boxing)

```
int testInt = 1;
Integer wrapInt = testInt;
wrapInt = new Integer(testInt);
testInt = wrapInt;
```

Casting (instaceof)

Explizit: mit cast-Operator (type), möglich bei einfachen, numerischen Typen; Referenztypen; Wrapper-Klassen dank Auto-Boxing.
Kann zu Informationsverlust führen.
Implizit: bei einfachen numerischen Datentypen → Typumwandlungen in einen "breiteren Typ".



Der **instanceof** Operator stellt fest ob ein Objekt von einer bestimmten Klasse ist, bzw. von einer davon abgeleiteten Klasse:

```
public class Foobar extends Test

Foobar foo = new Foobar();
if (foo instanceof Test) //true
```

Methoden

Überladen:
Mehrere Methoden mit gleichem Namen aber anderen Parametern.

Überschreiben:
Eine Unterklasse kann eine Methode der Oberklasse mit einer eigenen Implemetantion der Methode überschreiben.

Neu Definieren:
Eine `private` Methode der Oberklasse kann in der Unterkl. neu definiert werden.

Simulated Call-by-Reference

Java unterstützt grundsätzlich nur "**Call-by-Value**". Durch das übergeben einer Referenz auf ein Objekt kann dieses verändert werden (**Simulated-Call-by-Reference**)

```
MyTest tst = new MyTest();
MySim sim = new MySim();
test.changeValues(sim);
```

Klassenmethoden und -variablen

Werden mit dem **Schlüsselwort "static"** bezeichnet.

Gibt's nur einmal im Speicher (Method Area) und werden einmal initialisiert.

Diese Methoden und Variablen (Konstanten) sind auch ohne Objekte verfügbar. Aufruf:

```
Klasse.methode();
Klasse.WERT;
```

Klassen/Objekte

Konstruktor

Konstruktorname = Klassenname
Dient zur Initialisierung eines Objekts.

Default-Konstruktor wird vom Compiler zur Verfügung gestellt, falls dieser nicht selbstgeschrieben wird.
Es können mehrere Konstrukturen mit verschiedenen Parametern erstellt werden.

Aufruf eines Konstruktors im Konstruktor:
this() = Konstruktor der eigenen Klasse
super() = Konstruktor der Oberklasse

Muss der erste Aufruf im Konstruktor sein!

```
public Klasse() {
    super();
}
```

Garbage Collector (finalize())

Der Garage Collector liegt in der Kontrolle der virtuellen Maschine.

Er kann angefragt werden mit:

```
System.gc();
```

oder

```
Runtime.getRuntime().gc();
```

Wenn alle Referenzen auf ein Objekt "null" gesetzt wurden, dann wird das Objekt mit grösster Wahrscheinlichkeit vom GC entfernt, dies ist jedoch nicht garantiert.

finalze(): Wird ausgeführt wenn ein Object vom Garbage Collector entfernt wird.
Als letzter Aufruf in der finalize()-Methode super.finalize() aufrufen!

Reservierte Wörter

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

Enum / Switch

```
enum Form { RECHTECK, KREIS }
Form form = Form.RECHTECK;
switch (form) {
  case RECHTECK:
    break;
  case KREIS:
     break;
  default:
    //if no case matched.
}
```

Operatoren (nach Priorität)

`[]`	Array-Index
`()`	Methodenaufruf
`.`	Komponentenzugriff
`++ --`	Inkrement / Dekrement
`+ -`	Vorzeichen
`~`	bitweises Komplement
`!`	Logische Negation
`(type)`	Typ-Umwandlung
`new`	Erzeugung
`* / %`	Multiplikation, Division, Rest
`+ -`	Addition, Subtraktion
`+`	Stringverkettung
`<<`	Linksshift
`>>`	Vorzeichenbehafteter Rechtsshift
`>>>`	Vorzeichenloser Rechtsshift
`< <= > >=`	Vergleich kleiner (oder gleich), grösser (oder gleich)
`instanceof`	Typüberprüfung eines Objekts
`==`	Gleichheit
`!=`	Ungleichheit
`& &&`	bitweises / logisches UND
`^`	bitweises Exclusiv-ODER
`| |`	bitweises/logisches ODER
`? :`	Bedingung
`=`	Wertzuweisung
`*= /= %= += -= <<= >>= >>>= &= ^= |=`	Kombinierte Wertzuweisung

Side Effects:
In Java werden die Operanden eines Operators **strikt von links nach rechts ausgewertet**. Dies bedeutet, dass der Nebeneffekt des linken Operand vor der Bewertung des rechten Operand erfolgt ist.

Exception

```
public void blah() throws Exception {
  throw new Exception("WTF?");
}
public static void main(String[] args) {
 try {
   blah();
 } catch (Exception e) {
   System.out.println(e.getMessage());
 }
}
```

		Organisation	Freiheit beim Zugriff	Mechanismus beim Zugriff	Duplikate	Besonderheiten
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	LinkdedList	geordnet	sequenziell	nächtes 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	-

Collections aus Object-Elementen

Konzepte

Collections

Iterator
```
Iterator<Test> it =
tstSet.iterator();
while (it.hasNext()) {
   Test myTst = (Test) it.next();
   System.out.println(myTst);
}
```
Liste
```
List<A> l = new ArrayList<A>();
l.add((A)a); (A)l.get(0);
for (A a : l) { … }
```
Map
```
Map<Integer, String> m = new
HashMap<Integer, String>();
m.put(5, "fa"); (String)m.get(5);
for (String s : m.values()) { … }
```
Versionisierung
```
private static final long
serialVersionUID = 1;
```

Generics

Generische Klassen werden parametrisiert, d.h. Gewisse Datentypen werden nicht explizit verwendet sondern mit einem Platzhalter versehen:
```
public class GenKlasse <A, B> {
  private A wertA;
  private B wertB;
}
```
Beim Verwenden der Klasse:
```
Punkt<Integer> intPunkt = new
Punkt<Integer>(1,2);
```
Wildcards
?: nur für Referenzen auf beliebige Typen
<T extends Number>: Von Number abgeleitete Klassen
-Höhere Typsicherheit: Erkennen von Typ-Umwandlungsfehlern zur Kompilierzeit statt zur Laufzeit
-Wiederverwendbarkeit von Soucecode
- **Vermeiden von expliziten Casts** beim Auslesen ais eines

Interfaces

Eine Schnittstelle ist ein Entwurf, es legt den Zugriff von aussen fest. Eine Klasse kann ein Interface implementieren:
```
public class MyTest implements Test
```
Schnittstelle definieren:
```
interface Tester {
    public boolean test();
}
```
Wenn eine Klasse ein Interface implementiert müssen **alle darin festgelegten Methoden implementiert** werden. Ein Interface ist ein **Referenztyp**. Es können davon also Referenzvariablen gebildet werden die auf Objekte zeigen, deren Klassen das Interface implementieren.

Packages

```
package test.subtest.subsubtest;
```
Definition immer am Anfang der Klasse. Zu verwendende Klassen müssen public sein.
```
import test.subtest.*;
```
Klassen aus dem Package subtest erreichbar, nicht jedoch aus subsubtest!

Geschachtelte Klassen

Klassendefinitionen innerhalb der Klasse. (Gründe: Sichtbarkeit, Gruppierung, State Pattern)
- Statische innere Klasse: `class Out{static class In{}}`
- Mitgliedsklasse: `class Out{class In{}}`
- Lokale Klasse: `class Out{Out(){class In{}}}`
- Anonyme innere Klasse: `new Runnable(){public void run(){}}`

Abstracts

Es können **keine Objekte von abstrakten Klassen erstellt werden**, sie dienen als Vorlage und zwingen den Programmieren Unterklassen zu erstellen. Sobald eine Methode in der Klasse abstract ist, ist die ganze Klasse abstrakt. Abstrakte Klassen können konkrete Implementierungen von Methoden enthalten, welche so auch vererbt werden.
```
abstract class Gameobject{String
name;}
abstract boolean useOn(Gameobj
object);
```

Annotations

Werden als Meta-Kommentar zu einer Methode hinzugefügt:
```
@Override
public String toString() {
```
Fehlt die Methode in der Oberklasse, wird vom Compiler eine Fehlermeldung ausg.

Annotationen entsprechen Interfaces (der Compiler legt für jede Annoation ein von der Klasse "Annotation" abgeleitetes Interface an).
```
public @interface MyAnnotation
```
Wichtigste Annotationen:

Annoation	Bedeutung
@Override	Prüft ob Methode überschrieben wird
@Deprecated	Veraltete Methode
@SuppressWarnings	Compiler-Warnungen nicht anzeigen

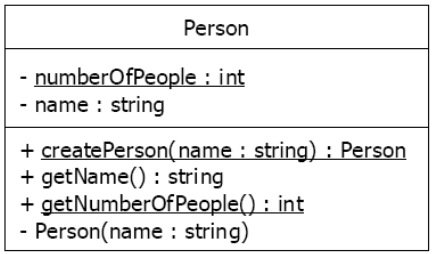
Annoationen sind vor allem wichtig beim Schreiben von Bibliotheken.

Modifikatoren

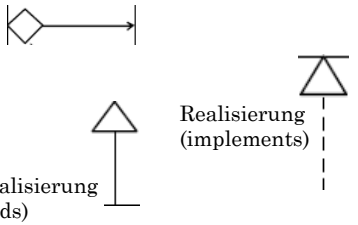
	Datenfeld	Methode	Konstruktor	Class	Interface
abstract		X		X	X
final	X	X		X	
native		X			
private	X	X	X		
protected	X	X	X		
pulic	X	X	X	X	X
static	X	X		X	X
synchronized		X			

UML

\+ = public - = private under = static
\# = protected ~ = package → = zugehörig
<<interface>> *AbstrakteKlasse*



Aggregation
(part of)



File Input/Output

Bitweise:
```
FileInputStream (Output)
BufferedInputStream (Output)
```
Character:
```
FileReader (-Writer)
BufferedReader (-Writer)

import java.io.*;
public class IOTest {
 public static void main(String[]
args) {
  try {
   readwrite();
  } catch (IOException e) {
   e.printStackTrace();
  }
 }
 private static void readwrite()
throws IOException {
  BufferedReader in = new
BufferedReader(new
FileReader("C:\\f1.txt"));
  BufferedWriter out = new
BufferedWriter(new
FileWriter("C:\\f2.txt"));

  int c;
  while ((c = in.read()) != -1) {
   out.write(c);
  }
  in.close();
  out.close();
 }
}
```