# Zusammenfassung OOP Februar 2019

Klassen und Objekte  
Klassen stellen uns als Vorlagen für unsere Objekte vor.   
Objekte sind konkrete Instanzen von Klassen.   
Alles was in einer Klasse definiert wurde, ist für ALLE Objekte verfügbar.   
Die Datensätze von Objekte sind dabei immer eigene.  
  
Erstellen von Objekten  
Wir erinnern uns an das Deklariern und Initialisieren von Variablen.

Datentyp Variablennamen = Wert;  
int meineZahl = 5;  
Analog dazu das Erstellen von Objekten.  
Klassenname Objektname = new Klassenname();  
Durchschnitt derDurschnitt = new Durchschnitt();

Definition von Methoden und Attributen  
Methoden - Was macht die Klasse / das Objekt?  
Funktioniert genau wie bisher, ein Unterschied ist zu beachten!   
Das bisher voren angesetzte static fällt weg.   
Methoden sind wie bisher mit und ohne Parametern sowie mit und ohne Rückgabetyp definierbar.  
   
Attribute - Was weiß die Klasse / das Objekt?  
Funktioniert genau wie bisher, ein Unterschied ist zu beachten!   
Das bisher voren angesetzte static fällt weg.   
Alle Datentypen sind wie bisher verfügbar. Regeln über die Zugreifbarkeit gelten unverändert!  
  
Zugriff auf Methoden  
Um auf Methoden aus einen Objekt zuzugreifen, schreiben wir

Objektname.Methodenname();  
derDurchschnitt.berechneDurchschnitt();

Die runden Klammern sind weiterhin beim Aufruf von Methoden nötig!  
  
Übergabe von Parametern  
Parameter sind die Informationen, die wir von außen in die Methode übergeben. Sie sind NUR während der Bearbeitung des entsprechend Aufrufs sichtbar!   
Hier gelten auch alle bisher bekannten Regeln!   
Die Übergabe ist auch über Klassen hinweg möglich!

Annahme von Rückgabewerten  
Auch hier gelten alle bisher bekannten Regeln unverändert.   
Anstelle von void schreiben wir unseren Rückgabetypen.

String gebeText() {  
 return "Rückgabe 2";  
}

Bei Aufruf der Methode haben wir die Möglichkeit den Rückgabewert aufzugreifen.

System.*out*.println(derDurchschnitt.gebeText());

Sichtbarkeiten  
Werden verwendet, um den Zugriff auf Daten und Methoden zu beschränken.

private  
Alles was private deklariert ist, ist von außerhalb der Klasse NICHT sichtbar.   
Wenn wir dennoch Zugriff wollen, müssen wir getter oder setter definieren.   
Attribute (unsere Daten) deklariern wir IMMER als private.  
  
public  
Alles was public deklariert ist, ist von außerhalb der Klasse SICHTBAR.   
Methoden, die von außen (von anderen Objekten aus) nutzbar sein sollen, werden als public deklariert.  
  
protected  
Gewährt Zugriff im Stammbaum (von erbenden Klassen) und im aktuellen Package.

Ohne Zugriffsmodifizierer  
Gewährt Zugriff im aktuellen Package

Getter und Setter  
Getter  
Wird verwendet, um ein privates Attribut auszulesen.   
Gibt also den Wert des Attributs (der Daten) zurück.  
Setter  
Wird verwendet, um ein privates Attribut zu verändern.   
Speichert also einen neuen Wert im Attribut ab.

Konstruktor  
Wird genutzt, um Objekte zu erzeugen.   
Kann genutzt werden, um Daten zu initialisieren.   
**MUSS** public sein und **MUSS** immer wie die Klasse heißen und **DARF** **KEINEN** Rückgabetypen haben.

public abstract class Arzt {  
 // Konstruktor   
 public Arzt() {  
 System.*out*.println("Ein neuer Arzt wurde angelegt!");  
 }  
}

Konstruktoren können wie andere Methoden auch überladen werden, sprich Parameter annehmen. Damit ist es möglich auch mehrere Konstruktoren zu definieren:

public abstract class Arzt {  
 // Attribute / Instanzvariablen  
 protected boolean arbeitetImKrankenhaus;  
 protected String vorName, nachName;  
  
 // Konstruktor - solle Name und Nachname annehemen  
 public Arzt(String vorName, String nachName) {  
 this.vorName = vorName;  
 this.nachName = nachName;  
 }  
  
 public Arzt() {  
 System.*out*.println("Ein neuer Arzt wurde angelegt!");  
 }  
}

In Vererbungsketten, muss immer der Konstruktor, der Klasse darüber aufgerufen werden, wenn dieser vom Standardkonstruktor (leerer / impliziter Konstruktor) abweicht. Konstruktoren der Klasse darüber werden mit super() aufgerufen. Hierbei gilt es auch auf die Parameterliste zu achten, diese muss passend sein.

public class Chirurg extends Arzt {  
 // Konstruktor  
 public Chirurg(String vorName, String nachName) {  
 super(vorName, nachName);  
 }  
}

Methodenaufruf mit Objektübergabe  
Wir können beim Aufruf von Methoden auch Objekte als Parameter übergeben.   
Dadurch können wir Abläufe schreiben, die dann auf jedes Objekt, egal wie es heißt, anwendbar sind.   
Wir sparen uns viel Schreibarbeit, zum Beispiel beim Ausgeben der Informationen über einen Kunden.   
Dabei wird in der Parameterliste der Klassenname gefolgt von einem frei wählbaren Paramenternamen geschrieben.

private void printCustomerInfo(Customer customer) { ... }

Wir haben dann innerhalb der Methode printCustomerInfo Zugriff auf das Objekt customer.

# ArrayList

Bietet uns die Möglichkeit während der Laufzeit in der Länge verändert zu werden. Im Gegensatz zum Array muss die ArrayList in der Länge nicht vordefiniert werden.  
Um sie zu nutzen müssen wir sie einbinden.

import java.util.ArrayList;

## Initialisieren

Um eine neue Liste zu initialisieren, schreiben wir:

ArrayList<Product> productList = new ArrayList<Product>();

Product wird durch die entsprechende Klasse oder einen Datentyp (int, String, ... ) ersetzt, productList durch den Namen unserer Liste.

## Neue Einträge ergänzen

Um auf die Liste zuzugreifen und neue Inhalte hinzuzufügen, nutzen wir die .add(Eintrag) Methode. Dabei übergeben wir ihr das Objekt / die Variable / den Wert, das / die / den wir aufnehmen wollen.

productList.add(product);

Einträge auslesen  
Um auf einen Eintrag an einer spezifischen Stelle zuzugreifen, ähnlich wie bei einem Array, nutzen wir .get(Index).

productList.get(0);

## Einträge entfernen

Um etwas zu entfernen, nutzen wir .remove(Index)

productList.remove(0);

Zu beachten gilt es, dass die Indizes der übrigen Einträge angepasst werden. Alle nachfolgenden rutschen um eins zurück, damit die Lücke, die durch das Entfernen entstand, geschlossen wird.

## Einträge an einer spezifischen Stelle ergänzen

Um etwas an einer spezifischen Stelle einzufügen, nutzen wir .add(Index, Eintrag)

productList.add(8, iPhone);

An der achten Stelle wird ein Objekt iPhone eingefügt. Zu beachten ist auch hier, die nachfolgenden Einträge rutschen um eins weiter. Die Indizes werden also auch hier angepasst.

# Vererbung

Wir können Klassen von einander erben lassen, also erweitern. Dabei werden alle Methoden und Attribute der Superklasse (auch Mutterklasse oder Oberklasse genannt), die die Sichtbarkeit public oder protected tragen, in die Subklasse (auch Kindklasse oder Unterklasse genannt), übertragen.

Eine gute Veranschaulichung, inklusive Klassendiagramm und Codebeispiel finden wir im Buch „Java von Kopf bis Fuß“ unter Kapitel 7 „Polymorphie und Vererbung“ auf Seite 168f.

## Polymorphie – Überschreiben

Beschreibt das **gleichzeitige** **Existieren** von **Methoden** mit dem **gleichen Namen** und der **gleichen Parameterliste** in der **Mutter**- sowie **Kindklasse**.

public class Arzt {  
 // Attribute / Instanzvariablen  
 protected boolean arbeitetImKrankenhaus;  
  
 // Methoden  
 public void patientBehandeln() {  
 System.*out*.println("Ich bin in einer Behandlung!");  
 }  
}

public class Chirurg extends Arzt {  
 // Attribute  
  
 // Methoden  
 @Override  
 public void patientBehandeln() {

System.*out*.println("Ich operiere!");  
 }  
}

**@Override** zeigt uns, dass wir eine **Methode** **überschreiben** wollen. Es ist **nicht** **notwendig**, aber hilfreich, da wir einen **Fehler** **bekommen**, sollten wir uns **vertippen** oder die **Parameterliste nicht korrekt** einsetzen.   
Außerdem macht es das Programm **übersichtlicher**, da wir sofort erkennen, dass wir eine andere Methode **überschreiben**, **ohne** dass wir die Klasse Arzt **untersuchen** müssen.  
Erstellen wir ein Objekt von **Chirurg**, so wird dieser „**Ich** **operiere!**“ beim Aufruf von patientBehandeln() ausgeben, ein **Arzt** jedoch nur „**Ich bin in einer Behandlung!**“.

## Überladen (Wiederholung)

Wenn Methoden mit **gleichem** Namen aber **unterschiedlichen** Parametern existieren, sprechen wir von überladenen Methoden.

public void patientBehandeln() {  
 System.*out*.println("Ich bin in einer Behandlung!");  
}  
  
public void patientBehandeln(String name) {  
 System.*out*.println("Ich behandle den Patienten " + name);  
}  
  
public void patientBehandeln(int anzahl) {  
 System.*out*.println("Ich behandle gerade " + anzahl + " Patienten.");  
}

Ausschlaggebend hierbei ist **nicht** der Name des Parameters sondern der **Datentyp**. Auch die **Reihenfolge** der Parameter macht einen Unterschied!

public void patientBehandeln(int zahl, String name) {  
}  
  
public void patientBehandeln(String name, int zahl) {  
}

Diese beiden Methoden sind **nicht** dieselben! Die Parameterliste ist unterschiedlich, einmal int und String, einmal String und int. Das ist **nicht** das gleiche!

## Abstrakte Klassen

Abstrakte Klassen dienen als **Vorlage** für davon erbende Klassen. Wir können **kein** Objekt von einer abstrakten Klasse erstellen. Sobald **eine** **abstrakte** **Methode** innerhalb einer Klasse vorhanden ist, **muss** diese als **abstrakt** **gekennzeichnet** werden. Abstrakte Methoden **müssen** in den Kindklassen durch **konkrete** **Methoden** **überschrieben** werden. Klassen **können** jedoch **auch** **ohne** abstrakte Methoden als abstrakt gekennzeichnet werden!  
Das ist hilfreich, um **obligatorische** (verpflichtende) **Eigenschaften** einer Klasse zu definieren, die aber bei jeder Kindklasse **unterschiedlich** sind.

public abstract class Arzt {  
 // Abstrakte Methoden  
 public abstract void patientBehandeln();  
}

# Interfaces

Zu deutsch, **Schnittstellen**, sind vergleichbar mit **komplett** **abstrakten** **Klassen**. Sie enthalten **KEINE** konkreten Methoden, **nur** **Methodenköpfe**, definieren also nur den **Rückgabetypen** sowie den **Methodennamen** und die **Parameter**, welche die Methode akzeptieren soll.   
Außerdem können wir nur **konstante** (final) und **statische** (static) Attribute, also **Konstanten** definieren. Es ist nicht nötig final und static auszuschreiben, das wird automatisch übernommen.

Interfaces bieten uns die Möglichkeit das „**was**“ und „**wie**“ zu **trennen**. Wir geben vor, was erwartet wird, wir definieren also Methoden, implementieren Sie aber **NICHT** im Interface.

In Java schreiben wir folgendes:

interface Omega {}

Wenn wir dieses einbinden wollen, nutzen wir implements:

class Alpha implements Omega{}

Implements steht nach extends, wenn zusätzlich noch eine Vererbung stattfindet. Im Gegensatz zur Vererbung, wo keine Mehrfachvererbung möglich ist, ist bei Interfaces das Einbinden von mehreren Interfaces möglich. Dazu listen wir mehrere Interfaces durch ein Komma getrennt nacheinander auf.

class Alpha extends Gamma implements Beta, Zeta, Omega{}

Innerhalb eines Interfaces können wir abstrakte Methoden(-köpfe) definieren:

interface Beta {  
 int getX();  
 int getY();  
}

Diese sind automatisch public und abstract, das müssen wir aber NICHT extra definieren.

interface Beta {  
 public abstract int getX();  
 public abstract int getY();  
}

Diese Methoden müssen wir in den Klassen, die das Interface verwenden ZWINGEND implementieren, ganz analog zu abstrakten Methoden aus abstrakten Klassen.

class Delta implements Beta {  
 public int getX() {  
 return 5;  
 }  
 public int getY() {  
 return 5;  
 }  
}

Attribute, die in einem Interface definiert werden, sind automatisch static und final, also Konstanten. Daher schreiben wir die Namen der Attribute in Großbuchstaben.

interface Zeta {  
 int *ZAHL* = 1;  
 String *WORT* = "Hallo";  
}

Auch hier müssen wir nicht extra erwähnen, dass es sich um public static final handelt:

interface Zeta {  
 public static final int *ZAHL* = 1;  
 public static final String *WORT* = "Hallo";  
}

Der Zugriff auf diese Attribute erfolgt mit InterfaceName.AttributName, hier also Zeta.ZAHL oder Zeta.WORT:

class Delta implements Zeta {   
 public int getX() {  
 return Zeta.*ZAHL*;  
 }   
 public int getY() {  
 return 5;  
 }  
 public void helloWorld() {  
 System.*out*.println(Zeta.*WORT*);  
 }  
}