LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Javakurs für Fortgeschrittene

Einheit 01: Organisation, Wiederholung und File IO

Lorenz Schauer
Lehrstuhl für Mobile und Verteilte Systeme

Heutige Agenda

Organisatorisches

- Ziele und Aufbau
- Zielgruppe, Vergütung, Webseite
- Kontakt

Wiederholung von Grundlagen

- Klassen & Objekte
- Vererbung

Dateizugriff – File IO

Praxis:

- Konto
- Girokonto
- Log-File

Lernziele

- Kurze Wiederholung von Grundlagen und Warm-Up
- Umgang mit Dateien kennenlernen
- Dateien schreiben und lesen lernen

Organisatorisches zum Kurs

- **Voraussetzungen:**
  - Grundverständnis von objektorientierten Programmiersprachen
  - Grundlagen und praktische Programmierkenntnisse in Java werden vorausgesetzt
    - Siehe Javakurs für Anfänger aus dem WS 2016/17: http://www.mobile.ifi.lmu.de/lehrveranstaltungen/java-fuer-anfaenger-ws1617/

- **Ziele**:
  - Praktische Programmierkenntnisse in Java vertiefen und erweitern
  - Vermittlung von weiteren theoretische Grundlagen und praktischen Konzepte der Programmierung mit Java SE

- **Aufbau**:
  - Mischung aus theoretischen und praktischen Programmiereinheiten
  - Kleinere Programmieraufgaben müssen während der Veranstaltung selbstständig gelöst werden (ggf. mit Hilfestellung)
    - Bitte bringen Sie daher auch immer **Ihr eigenes Gerät** (Laptop) mit!

- **Zielgruppe:**
  - V.a. Bachelorstudenten mit Haupt- bzw. Nebenfach Informatik
  - Studenten mit Interesse an der Programmierung mit Java SE

- **Vergütung**:
  - Freiwillige Zusatzveranstaltung, daher keine Vergütung

- **Zeit und Ort:**
  - Ab 27.04.2017 (1. Vorlesungswoche) immer Donnerstags, von 18.00 -20.00 Uhr c.t.
  - Hauptgebäude, Geschwister-Scholl-Platz 1, Raum: E 004

- **Webseite**:
  - http://www.mobile.ifi.lmu.de/lehrveranstaltungen/javakurs-fuer-fortgeschrittene-sose17/

- **Anmeldung**:
  - Obligatorische Anmeldung zum Kurs über Uniworx (https://uniworx.ifi.lmu.de/)

- **Veranstalter**:
  - Lorenz Schauer (Wiss. Mitarbeiter)
    - Büro:
      - Lehrstuhl für Mobile und Verteilte Systeme
        Oettingenstraße 67, Raum U160
    - Sprechstunde:
      - Montags, 10 - 12 Uhr
      - Donnerstags, 14.00 - 16.00 Uhr
    - Kontakt:
      - Mail: lorenz.schauer@ifi.lmu.de
      - Tel.: 089-2180-9157
      - Web: http://www.mobile.ifi.lmu.de/team/lorenz-schauer/

Literaturhinweise (Kostenlos)

The Java™ Tutorials:
Oracle
Praktische Online-Tutorials mit vielen Lektionen und Beispielen.
Online (kostenlos): http://docs.oracle.com/javase/tutorial/

Christian Ullenboom:
Java ist auch eine Insel,
Rheinwerk Computing, ISBN 978-3-8362-1802-3.
Online (kostenlos): http://openbook.rheinwerk-verlag.de/javainsel/

Guido Krüger, Heiko Hansen:
Java-Programmierung Das Handbuch zu Java 8,
Die HTML-Ausgabe der 7. Auflage (Stand 2011) kann kostenlos heruntergeladen werden: http://javabuch.de/download.html

Themen, die wir im Kurs u.a. behandeln wollen:

- GUI-Programmierung
  - JavaFX / (Swing)
- Weitere Datenstrukturen
  - (verkettete) Listen, Stack, Schlangen (Queues), Hash-Tables, Bäume (Trees)
- Nebenläufigkeit
  - Threads, Conditions, Synchronisation, Monitore
- Design-Pattern
  - MVC, Singelton, Builder, Observer, …
- Algorithmen
  - Sortieren & Suchen, Rekursion, …
- Architekturen
  - Client/Server, P2P
- Ggf. Ausblick auf mobile Plattformen
  - JavaME, Android
- U.v.m.

Teil 1: Kurze Wiederholung

Objektorientierung, Vererbung

Zusammenhang zwischen Klassen und Objekte

Klasse:

- Wird vom Programmierer geschrieben, gespeichert und kompiliert
- Stellt ein Konzept bzw. Plan gleichartiger Objekte dar (Bsp.: Student)
  - Welche *Eigenschaften* (Attribute) haben die Objekte der Klasse (Bsp.: Name, Alter,…)
  - Welches *Verhalten* (Methoden) bieten die Objekte der Klasse (Bsp.: studieren, schlafen,…)
- Beschreibt dadurch einen Teil der Realität

Objekt (= Instanz einer Klasse):

- Wird beim Ausführen des Programms erzeugt und spätestens beim Beenden wieder verworfen
- Existiert nur im Speicher
- Wird nach dem vorgesehenen Konzept bzw. dem Plan seiner Klasse erstellt:
  - Bekommt Werte für seine Attribute (Bsp.: „Hansi “, 21,…)
  - Verhält sich wie in seiner Klasse definiert (Bsp.: studieren, schlafen …)

Grundstruktur einer Java-Klasse
Beispiel: Auto


```java
// Package Deklaration
package java.fuer.anfaenger;

// Import externe Klassen
import java.util.*;

// Klassendefinition
public class Auto{

    // Instanzvariablen
    private String name;
    private int preis;

    // Konstruktor
    public Auto(String name, int preis){
        this.name = name;
        this.preis = preis;
    }

    // Methoden
    public void fahren(String a, String b){
        System.out.println("Das Auto faehrt von "+a+" nach "+b);
    }

    public void bremsen(){
        System.out.println("Ich bremse!");
    }

}
```

Eigenschaften (Attribute)

Konstruktor

Fähigkeiten (Methoden)

Auto
- name : String - preis : int
+ fahren (a: String, b: String) + bremsen ()

Objekte

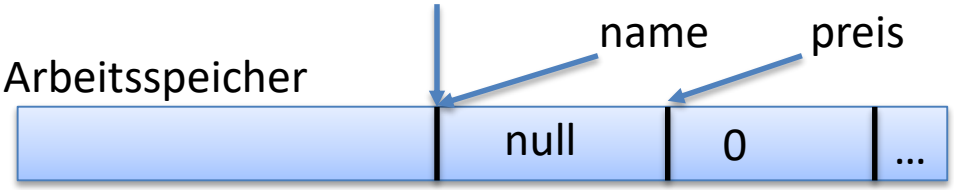
Ein Objekt ist eine eindeutige Instanz seiner Klasse

- Verfügt über dessen Attributen und Methoden
- Wird mit dem **new** Operator erzeugt, Bsp.:

```
Klassenname meinObjekt = new Klassenname();
```

- Besitzt:
  - *Zustand*: Aktuelle Werte der Instanzvariablen
  - *Verhalten*: Methoden
  - *Identität*: Eindeutige Speicheradresse

```
Auto mein_audi = new Auto();
```



Beispiel: Klasse Auto

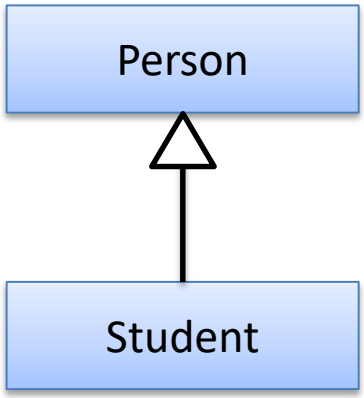
- Besitzt Instanzvariablen `String name` und `int preis`
- Objekt anlegen mit: `Auto mein_audi = new Auto();`
  - Erzeugt implizit Defaultwerte für den Namen und den Preis, nämlich `null` und `0`
  - Direkte Belegung der Attribute beim Erzeugen über den Konstruktor

Vererbung

Die **Vererbung** ist ein Kernprinzip der objektorientierten Programmierung

Motivation: Wiederverwendbarkeit von Klassen

- Neue Klassen müssen nicht immer komplett neu geschrieben werden!
- Oft reicht es aus, neue Klassen aus bestehenden abzuleiten => Vererbung
  - Neue Klasse übernimmt die Eigenschaften und das Verhalten der alten Klasse
    - Die neue Klasse wird auch als Sub-, Unter-, oder Kindklasse bezeichnet
    - Die alte Klasse wird auch als Super-, Ober-, Basis-, Wurzel-, oder Vaterklasse bezeichnet
  - Die Unterklasse kann weitere Attribute und Methoden definieren
  - Die Unterklasse kann Methoden der Oberklasse überschreiben oder erweitern
- Vererbung stellt eine „is-a“ Beziehung dar.
  - Ein Objekt der Unterklasse ist damit auch ein Objekt der Oberklasse
  - Bsp.: Ein Student ist auch eine Person



Programmieraufgabe „Warm-Up“

Erzeugen Sie ein neues Eclipse-Projekt „JavaFortgeschritten“ und darin ein neues Source-Package „uebung01“.

Erzeugen Sie darin eine neue Klasse „Konto“, die folgende Eigenschaften hat:

- `kontonummer: String`
- `kontostand: double`

Beide Attribute werden durch den entspr. Konstruktor gesetzt.

Zudem hat die Klasse folgende Methoden:

- Getter für kontonummer und kontostand
- `einzahlen(double betrag)`: erhöht den Kontostand um den Betrag
- `auszahlen(double betrag)`: verringert den Kontostand um den Betrag

Testen Sie Ihre Klasse mit einem Programm „Kontotest“, indem Sie ein Konto mit Kontonummer „00001“ und einem Kontostand von 1.000 Euro anlegen.

Lassen Sie sich zur Kontrolle Nummer und Kontostand ausgeben.

Zahlen Sie nun 500 Euro ein und heben anschließend 750,50 Euro ab.

Geben Sie wieder Kontonummer und Kontostand zur Kontrolle aus.

Programmieraufgabe 2: Vererbung

Leiten Sie nun von Ihrer Klasse „Konto“ die Subklasse „Girokonto“ ab, welche die geerbten Attribute um die Eigenschaft `limit: double` erweitert.

Das Attribut gibt das Kreditlimit des Kunden an und muss im neuen Konstruktor mit angegeben werden. Somit werden 3 Parameter für die Erzeugung eines Girokontos benötigt.

Die Klasse besitzt zudem noch einen Getter und Setter für das neue Attribut `limit`

Die geerbte Methode `auszahlen(double betrag)` soll nun so überschrieben werden, dass ein Betrag nur dann ausgezahlt wird, wenn nach dem Abzug das Kreditlimit nicht überschritten wurde. Ansonsten soll eine entspr. Fehlermeldung ausgegeben werden.

Testen Sie Ihre neuen Klasse wieder mit Ihrem Programm „Kontotest“ und legen Sie ein neues Girokonto „G-001“ mit einem Kontostand von 10 000 Euro und einem Limit von 1 000 Euro an.

Zahlen Sie nun 11 000 Euro aus, anschließend zahlen Sie 11 000 Euro ein und dann zahlen Sie erneut 11 001 Euro aus.

Lassen Sie sich bei jedem Schritt den aktuellen Kontostand ausgeben und kontrollieren Sie das Ergebnis.

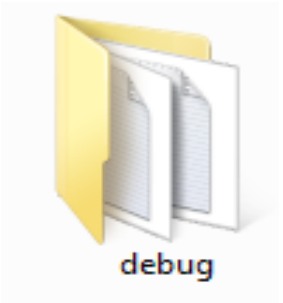
Teil 2: File IO

Umgang mit Dateien und Verzeichnissen

Umgang mit Dateien und Verzeichnissen

Um Daten persistent speichern zu können, müssen wir sie vom Programm auf einen Datenträger (Festplatte, USB-Stick, usw.) schreiben.

- Daten werden in Dateien organisiert, die sich in der jeweiligen Verzeichnisstruktur befinden.



Für die Arbeit mit Dateien und Verzeichnissen bietet Java die Klasse `java.io.File` an (seit Java 7 auch: `java.nio.file.Files` – wird später behandelt):

- Beim Erzeugen eines Objekts von `File` muss dem Konstruktor der Datei- bzw. der Verzeichnisname mitgegeben werden
  - `File(String path_or_name)` bzw. `File(String dir, String name)`
- Zudem bietet die Klasse einige wichtige Methoden:
  - `public boolean canRead() //Prüft Leserecht`
  - `public boolean canWrite() // Prüft Schreibrecht`
  - `public boolean createNewFile() // Erzeugt Datei, falls noch nicht vorhanden`
  - `public boolean delete() // Löscht die Datei bzw. das Verzeichnis`
  - `public boolean exists() // Prüft auf Existenz`
  - `public boolean renameTo(File dest) // Benennt die Datei/Verzeichnis um`

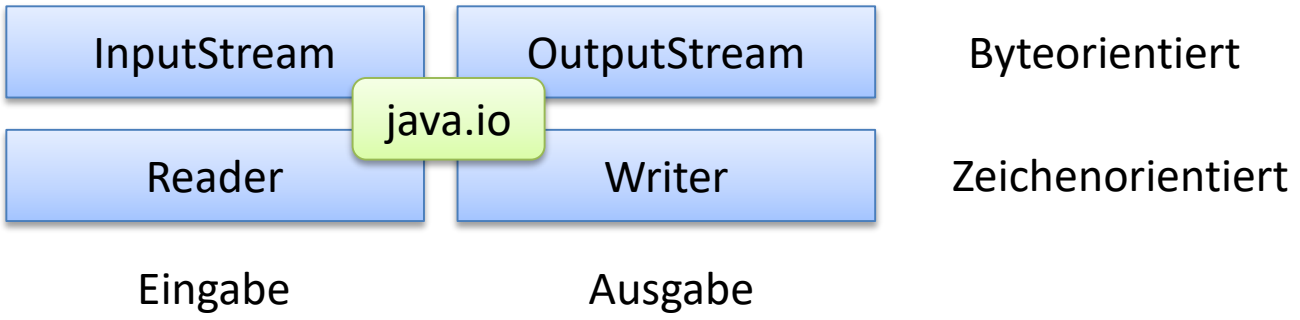
Ein- und Ausgaben auf Dateien

Grundlage für jede Ein- bzw. Ausgabe ist ein Datenstrom (Stream).

- Immer aus Sicht des Programms (Input, bzw. Output)
- Bereits kennengelernt bei:
  - Tastatureingaben mittels Standardeingabestrom `System.in`
    - Bsp.: `Scanner s = new Scanner(System.in);`
  - und Konsolenausgaben mittels Standardausgabe `System.out`
    - Bsp.: `System.out.println(String s);`

Nun wollen wir von Dateien lesen (Eingabestrom: `InputStream`) bzw. auf Dateien Schreiben (Ausgabestrom: `OutputStream`).

- Das Paket `java.io` bietet verschiedene Ein- und Ausgabeströme an:



Byteorientierte vs. Zeichenorientierte Datenströme

Byteorientierte Datenströme:

- Anfangs waren Datenströme generell byteorientiert
  - Byte ist kleinste direkt zugreifbare Informationseinheit
  - Vorteil: Übertragung unabhängig von der Art der Information (Text, Bild, Video, …)
- Alle Arten von Eingangsströme sind von `InputStream` abgeleitet:
  - `ByteArrayInputStream`, `FileInputStream`, `FilterInputStream`, usw.
- Alle Arten von Ausgangsströme sind von `OutputStream` abgeleitet:
  - `ByteArrayOutputStream`, `FileOutputStream`, `FilterOutputStream`, usw.

Zeichenorientierte Datenströme:

- Spezielle Datenströme, um Zeichenfolgen leichter übertragen zu können
  - Wird häufig fürs Lesen und Schreiben von Textdateien genutzt
  - Lesen und schreiben von Unicode-Zeichen vom Typ `char`
- Alle Arten von Eingangsströme sind von `Reader` abgeleitet:
  - `BufferedReader`, `StringReader`, `FilterReader`, `CharArrayReader`, usw.
- Alle Arten von Ausgangsströme sind von `Writer` abgeleitet:
  - `BufferedWriter`, `StringWriter`, `FilterWriter`, `CharArrayWriter`, usw.

Von Datei lesen

Um nun von einer Datei lesen zu können, muss diese zunächst über den Pfad angesprochen werden.

- `File my_file = new File("go/to/my/path/myFile.txt")`
  - Möglich, sollte aber vermieden werden, da nicht Plattformunabhängig!
  - Besser man nutzt das Trennzeichen abhängig vom OS mittels `File.seperator`
    - `new File("path"+File.separator+"myFile.txt");`

Zum Lesen aus einer Text-Datei erzeugen wir ein Objekt von `BufferedReader`:

- `BufferedReader reader = new BufferedReader(new FileReader(my_file));`
  - `throws IOException` (Überprüft, muss also behandelt werden!)
  - Verlangt einen Reader als Argument im Konstruktor.
    - In unserem Fall wäre das ein FileReader, der die entsprechende File als Argument bekommt:

Wir können den Inhalt nun mittels `readLine()` zeilenweise lesen:

- `String zeile = reader.readLine();`
  - Die Methode gibt `null` zurück, wenn kein Zeile vorhanden ist, also i.d.R. das Ende der Datei erreicht ist.

Von Datei lesen


```
// Beispiel: Aus einer Textdatei zeilenweise lesen:

      File myFile = new File("myTextDatei.txt");

      BufferedReader reader = null;

      // Falls Datei nicht vorhanden, dann erzeugen:
      if(!myFile.exists()){
         try {
            myFile.createNewFile();
         } catch (IOException e) {
            e.printStackTrace();
         }
      }

      String line;
      try{
         reader = new BufferedReader(new FileReader(myFile));

         while((line=reader.readLine())!=null){
            System.out.println(line);
         }
      }catch(IOException e){
         e.printStackTrace();
      }finally{
         if (reader!=null){
            try {
               reader.close();
            } catch (IOException e) {e.printStackTrace();}
         }
      }
```

Ein File-Objekt erzeugen

Eine Variable von `BufferedReader` anlegen

Entspr. Datei erzeugen, falls noch nicht vorhanden

Den `BufferedReader` mit entspr. `FileReader` instanziieren

Den Inhalt zeilenweise lesen bis zum Ende

…und auf der Konsole ausgeben

Nicht vergessen: Den Stream am Ende schließen!

Auf Datei schreiben

Zum Schreiben von Text auf eine Datei erzeugen wir ein Objekt von `BufferedWriter`:

- Analog zum `BufferedReader` wird ein `Writer` als Konstruktor-Argument benötigt (in unserem Fall ein `FileWriter`):
  - `BufferedWriter writer = new BufferedWriter(new FileWriter(my_file));`
  - `throws IOException` (muss behandelt werden!)
  - Dieser Aufruf wird später den Inhalt der Datei überschreiben, falls vorhanden.
  - Damit der Inhalt angehängt wird, muss `FileWriter(my_file, true)` aufgerufen werden!

Wir können nun einen String in die Datei mittels `write()` schreiben:

- `writer.write("Hallo\n");`

Beachte, jeden Stream am Ende wieder zu schließen!

- `reader.close();`
- `writer.close();`

Auf Datei schreiben


```
// Text in eine Datei schreiben (anhängen)

     BufferedWriter writer = null;

     if(!myFile.exists()){
        try {
           myFile.createNewFile();
        } catch (IOException e) {
           // TODO Auto-generated catch block
           e.printStackTrace();
        }
     }

     try {
        writer = new BufferedWriter(new FileWriter(myFile,true));
        writer.write(„Mein Text für die Datei.");
     } catch (IOException e) {
           e.printStackTrace();
     }
     finally{
        if (writer!=null){
           try {
              writer.close();
           } catch (IOException e) {e.printStackTrace();}
        }
     }
```

`BufferedWriter` deklarieren

Falls Datei nicht existiert, bitte erzeugen…

`BufferedWriter` mit entspr. `FileWriter` instanziieren. Inhalt wird angehängt!

Text als String in Datei schreiben

Nicht vergessen!

Programmieraufgabe: Log-Datei schreiben/lesen

Sie wollen nun bei Ihrem Bankkonto aus der vorherigen Aufgabe mitprotokollieren, welche Transaktionen (Ein- und Auszahlungen) getätigt werden.

Dazu erweitern Sie Ihr vorheriges Programm um folgende Eigenschaften:

- Es soll eine Log-Datei angelegt werden, in welche sämtliche Transaktionen geschrieben werden.
  - Wenn eine Einzahlung erfolgt, dann soll die Log-Datei folgenden Eintrag erhalten:
    - „Einzahlung auf Konto <Kontonummer>: <Betrag> Euro.“
  - Wenn eine Auszahlung erfolgt, dann soll die Log-Datei folgenden Eintrag erhalten:
    - „Auszahlung vom Konto <Kontonummer>: <Betrag> Euro.“
- Schreiben Sie außerdem eine Methode `void seeLogFile()`, welche Ihnen den Inhalt der Log-Datei auf der Konsole zeilenweise ausgibt.

Tätigen Sie nun einige Ein- und Auszahlungen und sehen Sie sich anschließend die Log-Datei durch Ihre neue Methode und auf Ihrem Dateisystem an.

Zusatz: Nutzen Sie das `java.time` Paket und versehen Sie jeden Log-Eintrag mit dem aktuellen Zeitstempel