LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR DATENBANKSYSTEME UND DATA MINING

Ein Schnelldurchgang durch Java

Große Projekte mit Java

Christian Reiner

16.10.2017

DBS

Basierend auf den Folien von Prof. Dr. Hans Jürgen Ohlbach

Grundlegende Strukturen

- Klassen, Interfaces
  zur Definition von Objekttypen

- Packages (Mengen von Klassen)
  um eine bestimmte Funktionalität zu realisieren
  (Bsp.: Server, Client)

- Libraries (Mengen von Packages)
  um umfangreiche Funktionalitäten zusammenzufassen

- Projekte (Klassen, Packages, Bibliotheken)
  Nicht in Java, sondern von der Entwicklungsumgebung

Namenskonventionen

Sobald mehr als eine Person mit einem Programm zu tun hat, ist es extrem wichtig, sich eine vernünftige Namensgebung der Bezeichner (engl. identifier) im Programm zu überlegen. Alle Namen sollten den Sinn des Bezeichners wiedergeben.

Wichtige Programmierrichtlinien:

- Generell alles auf Englisch halten.
- Camel-Case Bezeichner verwenden! Daher nicht get-Name(){} sondern getName(){}.
- Klassen-Bezeichner aus Substantiven, beginnend mit einem Großbuchstaben.
- Methoden-Bezeichner, wenn möglich Verben, beginnend mit Kleinbuchstaben
- Konstantennamen aus Großbuchstaben mit _ . Daher zb.: MIN_VALUE

Klassen

- Access-Modifier Klassenname (extends Oberklasse, implements Interfaces)
- Instanzvariablen (Attribute)
  Access-Modifier Typ Name;
- Klassenvariablen (static)
  static Access-Modifier Typ Name
- Konstruktoren
  Klassenname(Argumente) *(immer Typ, Name)*
- Instanzmethoden
  Access-Modifier Ergebnistyp Name(Argumente) {code}
- Klassenmethoden (static)
  static Access-Modifier Ergebnistyp Name(Argumente){code}
- Innere Klassen

Ein erstes Java-Programm

File HelloWorld.java

```
public class HelloWorld
{
        public static void main(String[] args){

        System.out.println( “Hello World“ );

        }
}
```

Klassenname: muss mit dem Filename übereinstimmen

Main Methode: wird beim Start des Programms aufgerufen

Enthält die Parameter, die beim Aufruf des Programms mit java eingegeben werden

Druckt die Zeichenkette

Compilieren mit: javac HelloWorld.java erzeugt: HelloWorld.class

Ausführen mit: java HelloWorld

Access Modifier

Access-Modifier / Environment	class	package	subclass	world
public	+	+	+	+
protected	+	+	+	-
no modifier	+	+	-	-
private	+	-	-	-

Wenn möglich „private“ Access-Modifier verwenden.

Setter und Getter

```
public class Punkt {
        private int x;
        private int y;

        public setX(int x){this.x = x;}
        public getX(){return x;}
}

Punkt p = new Punkt(...)
p.set(5);
int a = p.getX();
```

Warum Setter und Getter?

- Während der Programmentwicklung kann man zum Testen Zusatzcode in die Setter und Getter einfügen
  (z.B. als Zähler, wie oft darauf zugegriffen wird)

- Man kann die internen Datenstrukturen später noch ändern, ohne Auswirkungen auf den Rest des Programms, z.B. Übergang von kartesischen Koordinaten zu Polarkoordinaten

- Es gibt Programme, die andere Klassen anonym aufrufen (Java-Beans). Das geht nur mit Settern und Gettern.

- Optimierende Just-in-Time Java Compiler können die Setter und Getter wieder eliminieren, daher kein Zusatzaufwand

Interfaces

Interfaces sind im wesentlichen Listen von Methodenköpfen ohne deren Implementierung.

Klassen implementieren ein Interface, indem sie alle geforderten Methoden konkret implementieren.

Motivationsbeispiel:
Sie schreiben eine Sortiermethode **sort(List l)**.

Der Sortieralgorithmus braucht nur eine Methode **compareTo**, um zwei Objekte zu vergleichen. Was die Objekte sonst noch für Methoden haben, ist völlig egal.

Dazu definiert man ein Interface, welches nur die **compareTo** vorschreibt. Für alle Klassen, die dieses Interface implementieren, können jetzt die Listen mit **sort** sortiert werden.

Generics

Generics sind Klassen und Methoden mit **Typvariablen.**

Motivationsbeispiel:

Sie wollen eine Klasse schreiben, die Listen von Objekten verwaltet. Die Klasse soll Methoden haben zum Einfügen und Löschen von Elementen, und für den Zugriff auf Elemente.

Die Art der Elemente ist völlig egal, es soll aber immer nur **ein Typ** von Elementen sein, nicht gemischt, z.B.

List<Student> (nur Studenten) List<Dozent> (nur Dozenten)

Dazu definiert man eine generische Klasse

class List<E>

Mit der **Typvariable** E**.**

Jetzt kann man eine konkrete Liste mit z.B.

List<Student> studenten = new List<Student>();

aufbauen. In diese Liste kommen definitiv nur Studenten.

Primitive Datentypen

• int	(Ganze Zahlen)	Integer
• float, double	(Gleitpunktzahlen)	Float, Double
• boolean	(true, false)	Boolean
• char	(Zeichen in Unicode)	Character

Dazu noch

- Strings (Zeichenketten)
- Arrays (sind leider keine Objekte im Java-Sinn)

Primitive Datentypen können nicht für Typvariablen eingesetzt werden.

Collection und Map Datentypen

Für Sammlungen (collections) von Objekten gibt es das oberste Interface Collection<E>. Dazu gibt es etliche Subinterfaces und implementierende Klassen, insbesondere:

Listen	(List):	ArrayList, LinkedList
Mengen	(Set):	HashSet, TreeSet, LinkedHashSet
Schlangen	(Queue):	LinkedList, PriorityQueue

Dazu gibt es noch Assoziativspeicher (Map): HashMap, TreeMap,...

Bsp.:

```
Map<String,Integer> telefonnr = new HashMap<String,Integer>();
telefonnr.put(“Mayer“,2132432);
System.out.println(telefonnr.get(“Mayer“);
```

Design Patterns

Es hat sich herausgestellt, dass es in der Softwareentwicklung (wie z.B. auch in der Architektur) immer wiederkehrende typische Teilprobleme gibt, mit identischen oder wenigstens analogen Lösungen. Indem man diese typischen Teilprobleme

bennent (so dass man sie wiedererkennen kann)

Standardkonstrukte, oder wenigstens standardisierte Vorgehensweisen, zu ihrer Lösung bereitstellt,

vereinfacht und beschleunigt man die Softwareentwicklung deutlich.

Beispiel:

Iteratoren, die in standardisierter Weise für nahezu beliebige Collectiontypen funktionieren.

Fabrikmethoden

Idee: Objekterzeugung **nicht** durch Konstruktor, sondern durch **Methode**.

Analogie aus dem Alltag:

Man bestellt einen Mietwagen z.B. der Mittelklasse.

(In Java: Auto auto = sixt.getMietwagen(„Mittelklasse“))

Die Autovermietung entscheidet dann, welches konkrete Auto vermietet wird.

Das Factory Method Pattern kann eingesetzt werden, wenn:

Die aufrufende Klasse nicht weiß, welche konkrete Klasse instanziiert werden soll.

Die aufrufende Klasse nicht weiß, wie eine konkrete Klasse instanziiert werden muss.

Die aufrufende Klasse Informationen zur Erzeugung einer Klasse mitgibt, anhand dieser das Factory Method Pattern entscheidet, welche konkrete Klasse instanziiert wird.

Eine Klasse die Verantwortung für die konkret zu erzeugende Klasse an ihre Unterklassen delegieren möchte.

Observer

Einsatzbereich:

Ein Objekt A ändert seine Werte, andere Objekte B wollen das mitbekommen und darauf reagieren.

Methode:

B installiert einen **Observer (Listener)** in A. Bei Änderungen von A werden entsprechende Methoden des Observers von A aufgerufen, so dass B informiert werde kann.

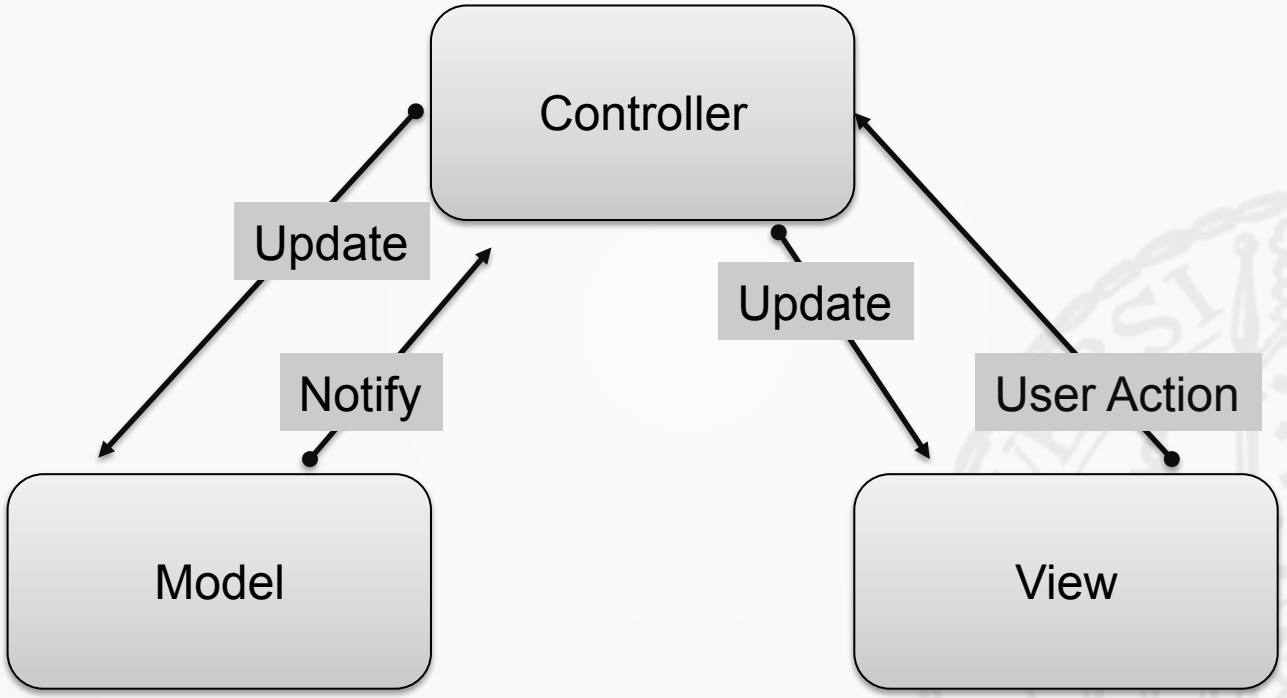
Beispiel 1:

Buttons einer graphischen Oberfläche. Das Drücken des Buttons bewirkt den Aufruf der installierten Observer.

Beispiel 2:

Ein Messwert, der sich über die Zeit verändert. Ein entsprechender Observer, der bei der Veränderung aufgerufen wird, aktualisiert eine graphische Anzeige.

Model View Controller (MVC)



Dokumentation mit Javadoc

```
/** Diese Klasse dient zur Berechnung des Umfangs eines Rechtecks
    @author C.Reiner
*/

public class Umfang {

        /**     Diese Methode dient nur zur Illustration der
                Parameterbehandlung durch javadoc.
                @param laenge = Parameter in int (Länge Rechteck)
                @param breite = Parameter in int (Breite Rechteck)
        */
        public static int umfang(int laenge, int breite){
        return 2*laenge + 2*breite;}}
```

UML – Unified Modeling Language