# Programmieren

# 1. First Things First

# 2.Nested Class

## 2.1 Local & Inner Class

In general each class has its own javaclass-File (or jar-File) (=toplevel-class). It is possible to define classes within another class.

Local classes are similar to inner classes:

Output:

…new OuterClass…

Constructor…x=99

…new Innerclass…

3.1 JTable Extension

3.1.1 Proposal concerning handling GUI – TableModel – Database

Class Diagram

Observableer

Observerer

GUI

JTable

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

Table Modle

Database

(Singleton)

## 3.1.2 Listener – Concept (e.g set cell value)

to inform GUI about exception because of invalid values

If setValue() throws Excetion, cursor remains in the cell.

Problem: setValue() cannot be overridden with “throws Exception”

* Implementing the listener concept

**Components**

1. Event Source

Generates events letting one or more listener know that something has happened. Therefore, a component must provide a mechanism to register and deregister event listener.

1. Listener Interface

Defines the methods used by an event source to dispatch events

1. Event Object

Holds all of the information necessary for a listener to figure out what happened

1. Listener

**EventObject:**

public class EventCellException extends EventObject{

private final String message;

public EventCellException(Object source) {

this(source, "not defined");

}

public EventCellException(Object source, String message) {

super(source);

this.message = message;

}

public String getMessage(){

return this.message;

}

}

## 3.1.3 Listener instead of Observer-Pattern

Observer and Observable is ok for a one to many relationship,

Meaning 1 Observable and many Observers watching the same event.

Observable

* 1 event
* 1 kind of Observer

#

#

It is quick, easy to do, and you don´t have a lot to worry about.

Observer

* Update (){many ifs (if many Observable a (o event}

Watching multiple Observable classes then things get messy.

Observer has one method that all Observable classes call.

You have to do a lot of checking to see which Observable raises which event.

Before you know how to handle it.

The biggest disadvantage to Observer and Observable, is that something must extend Observable

This is bad if the class you want to observe extends something else.

Eventsources and Listeners are not bound together.

EventSource

* Many Listener- Interfaces
* Many diff. Collections of Listener
* Many dir. Event objects

The events would be constructed (by Eventsource) as needed, and passed to the (appropriate NOT ALL) listeners.

Listener

* Many methods to react

Listener

You are not limited to extend Event like you are with Observable.

# 3. Swing (continued)

3.1.1 ComboBox in Cell

TableColumn tc = jTableLogBook.getColumnModel().getColumn(1);

JComboBox<Car> cboxCars = new JComboBox<>(db.getCars().toArray(new Car[0]));

Tc.setCellEditor(new DefaultCellEditor(cboxCars));

3.1.2 Editable LocalDate in Cell

@Override

Public Class <?> getColumnClass(int c){

Class<?> retValue = “x”.getClass();

//do not do it with date-type

//because cell would be not editable

If(getValueAt(0, c) != null && c > 0){

retValue = getValueAt(0,c).getClass();

}

Return retValue;

}

## 4.2 anonymous inner class

Idea: You can use them if you need to use a local class only once

Has no classname:

new ClassOrInterface() { … //implementation …}

* If new name\_of\_class

THEN anonymous Sublclass of classname.

* IF new name\_of\_interface

THEN anonymous implementation of interface AND inheritance from class Object.

* In any case are not allowed: extends and implements

Using anonymous classes can cause complexity

a/o restrictions (eg. No constructors, super(), this)

so avoid using them.

## 4.3 Lambda Expression

Overview

* Lambda provide a concise way to represent one method interface using an expression.
* Eg. Interface ActionListener with method actionPerformed() interface Runnable with run()
* They also improve the Collection libraries making it easier (?) to iterate through, filter, and extract data from a Collection

**Example Interface ActionListener**

ActionListener al = new ActionListener(){

@override

Public void actionPerformed(java.awt.event.ActionEvent evt){

System.our.println(“====new Al”);

}

};

* **Lambda expression**

ActionListener al = (java.awt.event.ActionEvent evt1) -> {

System.our.println(“===new al”);

}

**Example Interface Comparable**

TreeSet<Car> collCars = new TreeSet<>();

collCars.add(new Car(11, “Ford Etsel”);

collCars.add(new Car(22, “Ford Zwi”);

collCars.add(new Car(33, “Ford Dri”);

Collections.sort(collCars, new Comparator<Car>(){

@Override

Public int compare(Car c1, Car c2){

Return c1.getName().compareto(c2.getName());

}

});

For(Car c: collCars){

System.out.println(c);

}

**Lambda:**

Collections.sort(collCars, (Car c1, Car c2) -> c1.getName().compareTo(c2.getName()));

// (every object has an stream method)

collCars.stream().forEach((c) -> {

System.out.println(c);

}

**Resume:**

Examples on the original Oracle Website are not representative examples:

* One bonus should be avoiding of 3-5 code lines. This looks like much if the effective code consists of only 1 line, but it is nothing, if you have much more complex coding.
* Examples looks like as if intentionally coded in a semiprof. Manner to get the lambda-expression in a better light (e.g. ActionListener in a separate class (instead of doing it in the JFrame), Filtering in a separate class (instead of doing it in the Person with isOldie())
* You lose information e.g.

Predicate<Person> allDrivers = p -> p.getAge() >= 16;

Implicates knowledge of the interface Predicate with method test() which return a Boolean.

* Method-chaining not really redable, eg.

List<Person> p1 = Person.createShortList();

SearchCriteria search = SearchCriteria.getInsance();

List<Person> pilotList ? p1

.stream

.filter(search.getCriteria(“allPilots

* So if IDE-tools convert it automatically to lambda-expression, then ok.

4.3.1 Cell-Rendering

**October 30, 2017**

# 4 Pattern MVC (model – view – controller)

## 4.1 Overview

An application delivers one or more views of information.

These views should be synchronized immediately after each change of information.

Additional feature: the part of app, that is responsible for storing the information, can be implemented and run on a different platform.

e.g. view is running on browser (HTML), rest of app is running on webserver.

This pattern is an extension of the Observer Pattern (or Listener Concept).

Parts:

* Model…internal data representation and business logic (relationship of the data (constraints/functions/algorithms)
* View…representation for user
* Controller…eventhandler (event is fired (mostly) by user (e.g. button click in view), controller calls appropriate methods from model to cause a changing of data.

**November 6, 2017**

## 4.2 Structure

**Collaboration:**

* Each view has an appropriate controller (y/c pair), a model could service many v/c – pairs.
* The model informs all its listeners (views (and if necessary controllers)), which call appropriate getter.
* Controller needs (perhaps also generates) instances of model and views and
* Registers views as listeners.

Class Diagramm

Java no direct MVC support, but eg. With the help of Observer-Support we can implement.

## 4.3 MVVM (model – view – viewmodel)

* Is a variation of the MVC – pattern
* Better handling of modern UI – platforms (like MS – WPF, JavaFX, Android, HTML5)
* Model --- contains all of the information (like MVC)
* View … Layout ((nearly) no “codebehing”)
* ViewModel … contains th UI – logic

# 5 JavaFX

Databinding

ViewModel

Calls (business logic) mehtods and recives (updated) information

abc

Model

View

View Model

linkTo TXfl = „abc“

## 5.1 Introduction

Oracle states that JavaFX is replacing Swing as the new client UI library for Java.

**Features**

* New components like charts, media player.
* New language called FXML, which is used only to define the interface of an application, keeping it completely separate from the code logic.
* Scene Builder for generating FXML docs
* Control design with CSS-like syntax
* 🡪 MVVM

5.2 Architecture – Overview

The 3 main objects that get the program on the screen are (from outer to inner):

Stage – this is the outer shell of the applic ation and contains the entire program  
 Scene – whte switching to antoher view you have the satge change to a different sce

Scene

Stage

Root Pane

* Root pane can be any one of a number of JavaFx containers
* Examples: StackPane, gridPane, flowPane, borderPane

Stage – this is the outer shell of the application and contains the entire program

Scene – when switching to another view you have the stage change to a different scene

Root Pane/Container – holds all the parts of the application, like buttons, labels, textfields, etc. The root container can be represented by a number of javaFX containers such as the StackPane, GridPane, BorderPane and FlowPane

## Fill comboBox

ObservableList<Profile> tmpList = FXCollections.observableArrayList(Database.getTsPfofile());

//maybe runtime error 🡪 (List<Profile>) Databse.getTsProfile()

cboxProfile.setItems(tmpList);

cboxProfile.setValue(tmpList.get(0)); //to show 1st entry

## Passing Parameters

8. Communication between Activities

Intens are parts of a mechanism for exchanging data or messages between components.

## 8.1 New Activity

* Each new activity has to be known