Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
18. Dezember 2017

9. Übung zur Vorlesung
# Einführung in die Programmierung

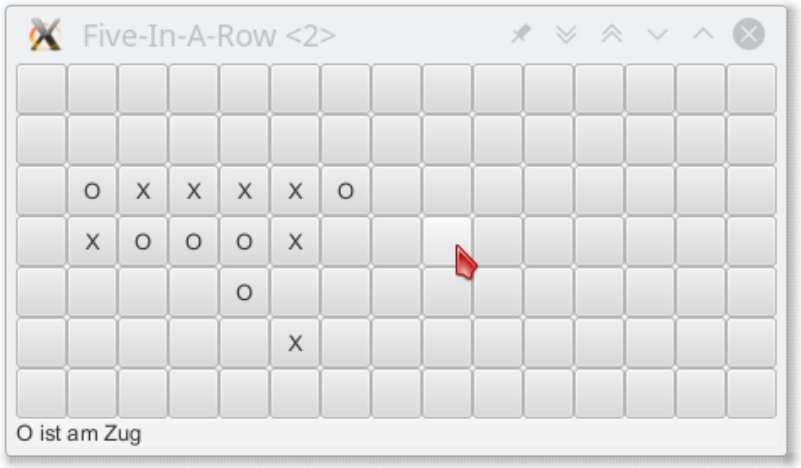
**A9-1** ***Catch & Throw*** Gegeben sei folgendes (nicht sehr sinnvolles) Java-Programm:

```
class PayloadA extends Throwable {
  private double x;
  PayloadA(double x)  { this.x = x; }
  public double get() { return x;   }
}

class PayloadB extends Throwable {
  private int x;
  PayloadB(int x)     { this.x = x; }
  public int get()    { return x;   }
}

class ThrowAndCatch {
  public static void main(String[] args) {
    try {
      f();
    } catch (PayloadB p) {
      System.out.println("Got payload B " + p.get());
  }  }

  static void f() throws PayloadB {
    try {
      g();
    } finally {
      System.out.println("Greetings from f's finally");
      throw new PayloadB(42);
  } }


  static void g() throws PayloadA {
    try {
      throw new PayloadA(4711);
    } finally {
      System.out.println("Greetings from g's finally");
} } }
```

a) Erklären Sie genau, wie das Programm abläuft. Machen Sie dies, ohne das Programm auszuführen! Welche Ausgabe wird produziert?

b) Warum muss in der Signatur von Methode `f` kein `throws PayloadA` deklariert werden, obwohl die darin aufgerufene Methode `g` sehr wohl ein `PayloadA`-Objekt werfen kann?

**A9-2** ***Fünf in einer Reihe*** Implementieren Sie eine kleine JavaFX-Applikation, mit der man das Spiel "Fünf in einer Reihe" spielen kann. Zwei Spieler, `X` und `O`, setzen abwechselnd, beginnend mit `X`, auf ein rechteckiges Spielfeld beliebiger Größe (meist 15x15 oder 19x19, wir wollen hier aber beliebige Rechtecke erlauben, deren Kanten größer als 5 sind). Der Spieler, der zuerst 5 horizontal, vertikal, oder diagonal zusammenhängende Steine gesetzt hat, hat gewonnen.




Unser Fenster besteht aus einer `GridPane` (wer möchte auch `FlowPlane` oder `BorderPane`), welche zwei Element übereinander anordnet. Das obere stellt das Spielfeld dar, während das untere Element ein einfaches `Label` ist, welches den Spielstatus anzeigen soll.

Das Spielfeld ist erneut eine `GridPane`, in der wir jede einzelne Zelle direkt durch Objekte der Klasse `javafx.scene.control.Button` darstellen. Dies ist zwar nicht wirklich hübsch, aber für unseren derzeitigen Kenntnisstand besonders einfach:

- Die Methode `void setText(String text)` erlaubt es uns, einen Text wie `X` oder `O` auf dem Knopf darzustellen.
- Die Methoden `void setMinWidth(double value)`, `void setMinHeight(double value)`, `void setMaxWidth(double value)`, `void setMaxHeight(double value)` erlauben es uns, die Größe der Knöpfe zu beeinflussen. Wir setzen diese einfach auf einen hinreichend großen Wert, damit alle Knöpfe gleich groß dargestellt werden.
- Die Methode `void setOnAction(EventHandler<ActionEvent> value)` erlaubt es uns, auf einen Druck des Knopfes zu reagieren. Am einfachsten machen wir dies mit einem Lambda-Ausdruck:

```
int x; int y; GridPane pane;
...
Button button = new Button();
pane.add(button,x,y);
button.setOnAction(event -> { System.out.println("Button("+x+","+y")"); });
```

Die Text Ausgabe in diesem Beispiel ist natürlich durch sinnvollen Code zu ersetzen. Das Beispiel zeigt aber, dass der Behandler sein Argument `event` hier gar nicht beachten

braucht, wenn wir die Position gleich beim Erstellen des Knopfes fest in den Behandler hineinschreiben.

Bei der Implementierungen folgen wir erneut dem MVC-Pattern. Wir können auch viele Teile aus dem zuvor behandelten Beispiel „Game of Life“ wiederverwenden, da wir erneut ein 2D-Spielfeld mit quadratischen Zellen haben. Allerdings ist es bei einer grafischen Benutzeroberfläche oft etwas schwierig, Controller und View ordentlich zu trennen.

Wir empfehlen hier, zuerst mit dem Modell anzufangen. Dieses sollte von beiden unabhängig sein und alle benötigten Methoden zur Verfügung stellen. Einige Klassen des Modells sollten vermutlich Unterklassen von `Observable` sein, und an geeigneten Stellen die geerbten Methoden `setChanged()` und `notifyObservers()` aufzurufen, damit sich die View bei Bedarf selbst aktualisiert.

Der Konstruktor der View erstellt die Elemente der Szene, setzt Ereignis-Behandler auf entsprechenden Aufrufe im Modell und hängt umgekehrt die Beobachter des Modells ein.

a) Überlegen Sie sich zuerst, welche Klassen sie erstellen sollten und welche Funktionalität diese bieten sollten.

b) Vergleichen Sie Ihren Entwurf mit der beiliegenden Dateivorlage. Unsere Dateivorlage ist nur ein Vorschlag! Trauen Sie sich ruhig, eigene Wege zu gehen!

c) Vervollständigen Sie nun Ihre Implementierung in folgenden Schritten:

   i) Initalisierung eines Spielfeldes mit fester Größe (einstellbar über eine Konstante, wie im Game of Life).

   ii) Ein Drücken eines Knopfes ändert des Beschriftung zu $\mathcal{X}$.

   iii) Abwechselnd werden $\mathcal{X}$ und `O` gesetzt. Die Statuszeile zeigt an, wer dran ist.

   iv) Das Drücken von bereits besetzten Spielfeldern wird ignoriert.

   v) Nach jedem Zug wird die Gewinnbedingung geprüft und ggf. in der Statuszeile angezeigt.

   vi) Ein Klick auf einem beliebigen Feld startet das Spiel neu, falls dieses beendet war. Das Spielfeld soll nun wieder leer sein, und $\mathcal{X}$ ist am Zug.

   vii) Eingabe von Breite und Höhe des Spielfelds über JavaFX vor Spielbeginn (z.B. über `javafx.scene.control.TextInputDialog` oder über Elemente im Hauptfenster).

   *Hinweis:* Bitte dazu nicht den Eingabe-Dialog von Folie 4.57ff verwenden, da dieser ein anderes GUI Framework verwendete und es sich nicht empfiehlt, mehrere Frameworks zu mischen.

**H9-1** ***Ausnahmen*** (4 Punkte; Alle `.java`-Dateien Ihrer Lösung abgeben)

a) Schreiben Sie eine Klasse `Neighbor` mit Prozedur `boolean neighbor(Object[] arr)`, die prüft, ob in einem `Object[]`-Array zwei benachbarte Elemente gleich sind. Die Prozedur soll also genau dann `true` zurückgeben, wenn `arr[i].equals(arr[i+1])` für wenigstens ein `i` gilt. Die `main`-Methode testet `neighbor` an den Kommandozeilenargumenten.

```
public class Neighbor {

  public static boolean neighbor(Object[] arr) {
  }

  public static void main (String[] args) {
    if (neighbor(args))
         System.out.println("Neighbor(s) in arguments!");
    else System.out.println("No neighbors in arguments!");
  }
}
```

b) Kopieren Sie `Neighbor` nach `NeighborWithException`, welches Sie wie folgt modifizieren: Schreiben Sie eine Ausnahme `ArrayTooShortException`, die ausgelöst werden soll, wenn das Array weniger als zwei Elemente enthält und insofern eine sinnvolle Prüfung auf benachbarte Elemente nicht möglich ist. Modifizieren Sie die Prozedur `neighbor` so, dass die Ausnahme ausgelöst wird, wenn das Eingabearray nicht lang genug ist. Ändern Sie `main`, so dass die Nachricht

```
Too few arguments to find neighbors!
```

ausgegeben wird, falls die Ausnahme ausgelöst wurde.

**H9-2** ***Mine Sweeper*** (8 Punkte; Abgabe: `Minesweeper.java` als Hauptklasse, plus ggf. weitere Klassen)

In dieser Aufgabe sollen Sie eine (vereinfachte) Variante des Spiel "Minesweeper" mit Hilfe von JavaFX implementieren.

Das Spielfeld ist erneut ein rechteckiges Gitter aus quadratischen Zellen. In jeder Zelle kann eine Mine stecken. Jede nicht verminte Zelle enthält eine Ziffer, nämlich die Anzahl der Minen innerhalb ihrer acht Nachbarn (am Rand sind etwas natürlich weniger). Am Anfang wählt der Computer die Position der Minen zufällig und zeigt das Spielfeld verdeckt. Der Spieler kann nun durch Linksklicks eine Zelle aufdecken. Liegt darunter eine Mine, hat er verloren. Anderfalls wird die Zelle mit der entsprechenden Zahl angezeigt — die Anzeige der Ziffer Null wird aber unterdrückt. Ist die Zahl Null, deckt der Computer auch alle Nachbarn auf. Wird dabei eine weitere Null aufgedeckt, werden auch deren Nachbarn vom Computer wieder aufdeckt. Diese letzter Vorgang wird so lange wie möglich wiederholt. Danach ist der Spieler wieder am Zug. Der Spieler gewinnt das Spiel, falls alle verdeckten Zellen nur noch Minen enthalten. Sobald der Spieler gewonnen oder verloren hat, soll das Programm eine entsprechende Meldung ausgeben und terminieren. Als Hilfe kann der Spieler verdeckte Zellen mit einem rechten Mausklick markieren und diese Markierung auf gleichem Wege auch wieder aufheben. Die Markierung kann der Spieler benutzen, um sich zu merken, dass eine Zelle vermint ist — die Markierung hat sonst keinen Einfluss auf den Spielverlauf. Beachten Sie das *Model-View-Control*-Entwurfsprinzip!

Das Fenster könnte wie folgt aussehen:


Dazu wurde die Zelle links unten mit der linken Maustaste angeklickt und die Zellen, die nun mit `P` markiert sind, mit der rechten Maustaste angeklickt.

**Wichtig:** Jede Klasse muss am Anfang einen Javadoc-Kommentar enthalten, welcher die Funktionsweise der Klasse in wenigen und einfachen Sätzen beschreibt!

**Hinweise:**

- Sie können große Teile des Lösungsvorschlags zu Aufgabe A9-2 wiederverwenden. Die Zellen können erneut durch `Button` dargestellt werden.

  Eine "aufgdeckte" Zelle kann einfach durch Deaktivierung des Buttons mit dem Aufruf `button.setDisabled(true);` realisiert werden.

  `setOnAction` reagiert nur auf Links-Klicks; stattdessen können Sie z.B. ähnlich wie hier vorgehen:

```
button.setOnMousePressed(event -> {
  if (event.isPrimaryButtonDown()) {
    System.out.println("Links-Klick");
  }
  if (event.isSecondaryButtonDown()) {
    System.out.println("Rechts-Klick");
  }
});
```

- Zufallszahlen können mit der Klasse `Random` bzw. mit deren Methode `nextInt` erzeugen.
- Im Gegensatz zu anderen Implementierungen müssen die Zahlen nicht farbig dargestellt werden, und die Flaggen und Minen können durch einen Text dargestellt werden.
- Für das Modell (im Sinne des MVC-Prinzips) kann es sinnvoll sein, die Zahlen der benachbarten Minen im Vorfeld zu berechnen und für jede Zelle einen Eintrag mit den relevanten Daten zu halten – inklusive der Information, ob sie verdeckt, markiert oder aufdeckt ist.
- Deklarieren Sie notwendige Konstanten wie die Größe des Spielfeldes (im Beispiel 20 auf 10) und die Anzahl der Minen (im Beispiel 10) explizit mit `static final`. *Ausnahme:* Die Werte werden über ein Eingabefenster, Konsole oder Kommandozeilenargumente eingelesen. (Hat keinen Einfluss auf Punktwertung.)

**Abgabe:** Lösungen zu den Hausaufgaben können bis Sonntag, den **7.1.18**, mit UniWorX nur als `.zip` abgegeben werden. Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (`www.tcs.ifi.lmu.de/lehre/ws-2017-18/eip/`).