Rules of the game

- Every group (2 – 3 students) must solve the **3 problems** individually.
- Each group has to present at least one problem.
- **Present the following:**
    1. Present and describe the problem.
    2. Discuss possible ways to solve the problem (discuss the theoretical base for your solution).
    3. Present the way you chose to solve the problem and defend it.
    4. Demonstrate and explain your code.
    5. Highlight the specialties in your solution and discuss strengths and weaknesses.
    6. Be prepared to answer questions.

Remark:
The presentation is a **team work**. Each team member presents approximately the **same amount** of information.

1 Auswertung von Lottoscheinen

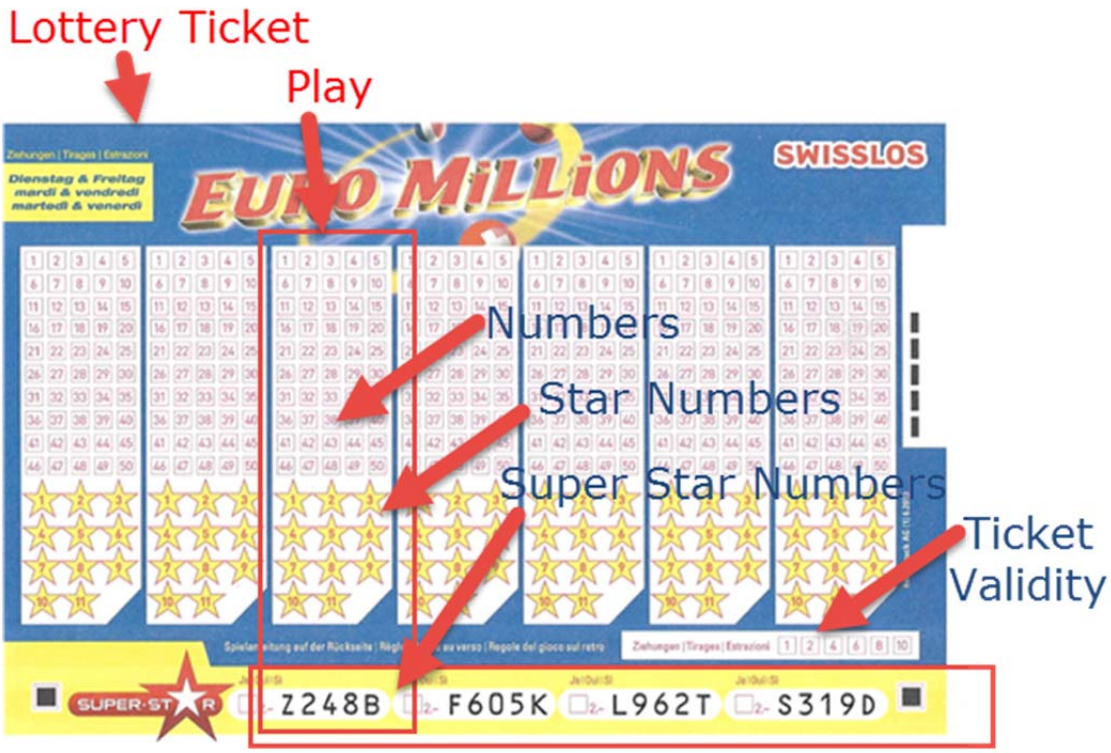
Sie kennen folgende Aufgabenstellung schon aus der Klausur.

Sie werden von einer Lottogesellschaft angefragt, eine Anwendung für die Erfassung und Übermittlung der verkauften Lottoscheine zu entwickeln.

1.1 Rahmenbedingungen

Die Ziehungen der Lottozahlen erfolgen **zwei Mal wöchentlich**, am Dienstag und am Freitag. Das jeweilige **Ziehungsdatum** ist wichtig für Ihre Anwendung.

Ein Lottoschein ist folgendermassen strukturiert:



Pro Schein (Ticket) können maximal **7 Zahlenpakete (Plays)** erfasst werden. Ein **leerer** Schein wird **nicht verarbeitet**. Nur richtig ausgefüllte Scheine werden verarbeitet.
Ein Zahlenpaket besteht aus einer eindeutigen Auswahl von **5 Zahlen** (*Numbers*) aus dem **Intervall 1..50**. Eine Zahl kann **nicht** doppelt gezogen werden.
Zudem müssen **zwei unterschiedliche weitere** Zahlen (Sterne: *Star Numbers*) aus dem **Intervall 1..11** gezogen werden.
Ein ausgefüllter Lottoschein kann für **mehrere Ziehungen** gültig gemacht werden.
Dazu muss in eindeutiger Art die Dauer der Ziehungen (*Ticket Validity*) für den Schein ausgewählt werden: Zur Auswahl stehen **1,2,4,6,8 und 10 Ziehungen**. Auch hier kann **nur eine** Option angewählt werden.
Optional können noch sogenannte *SuperStars* (*Super Star Numbers*) angewählt werden. Hierfür stehen **4 Vorschläge zur Auswahl**. Es können **beliebig viele aus den 4** angewählt werden.
Jeder Lottoschein ist einzigartig und somit **eineindeutig identifizierbar**.

Folgende Figur zeigt, welche Angaben zur Auswertung der Scheine Verwendung finden:

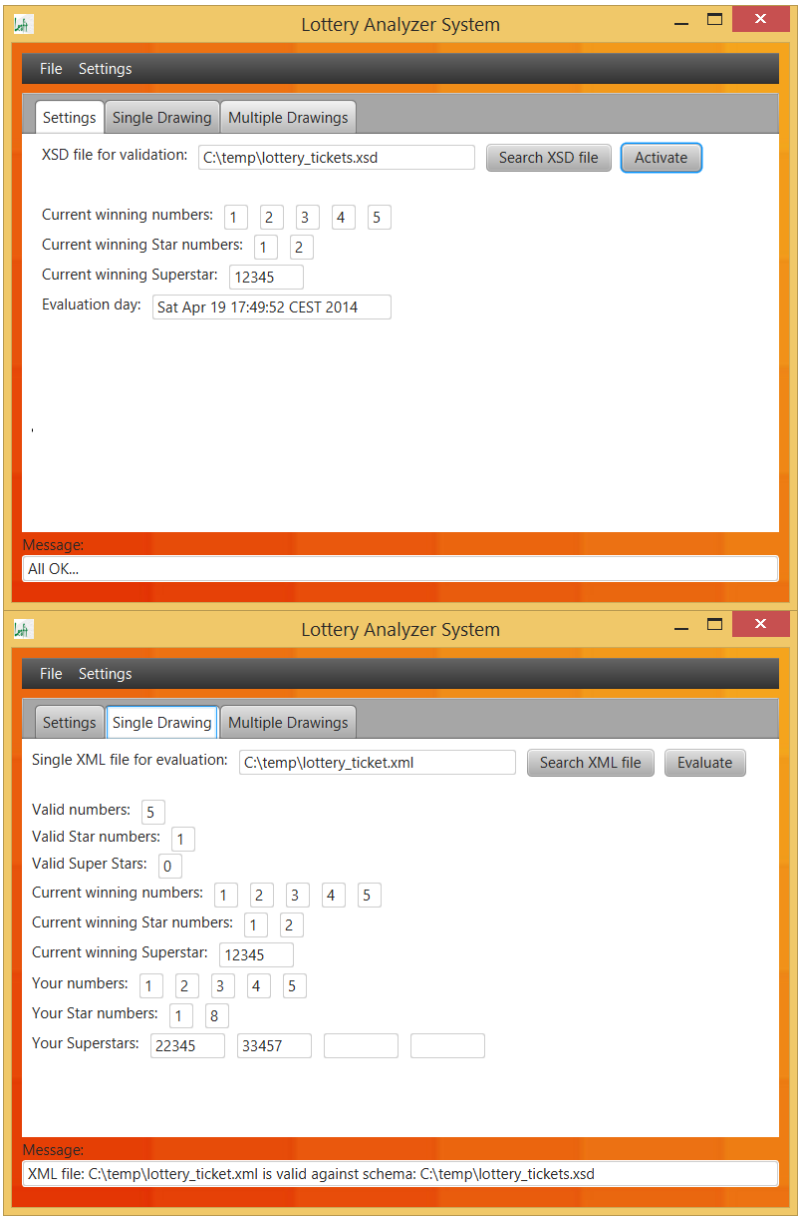
1.2 Ihre Aufgabe

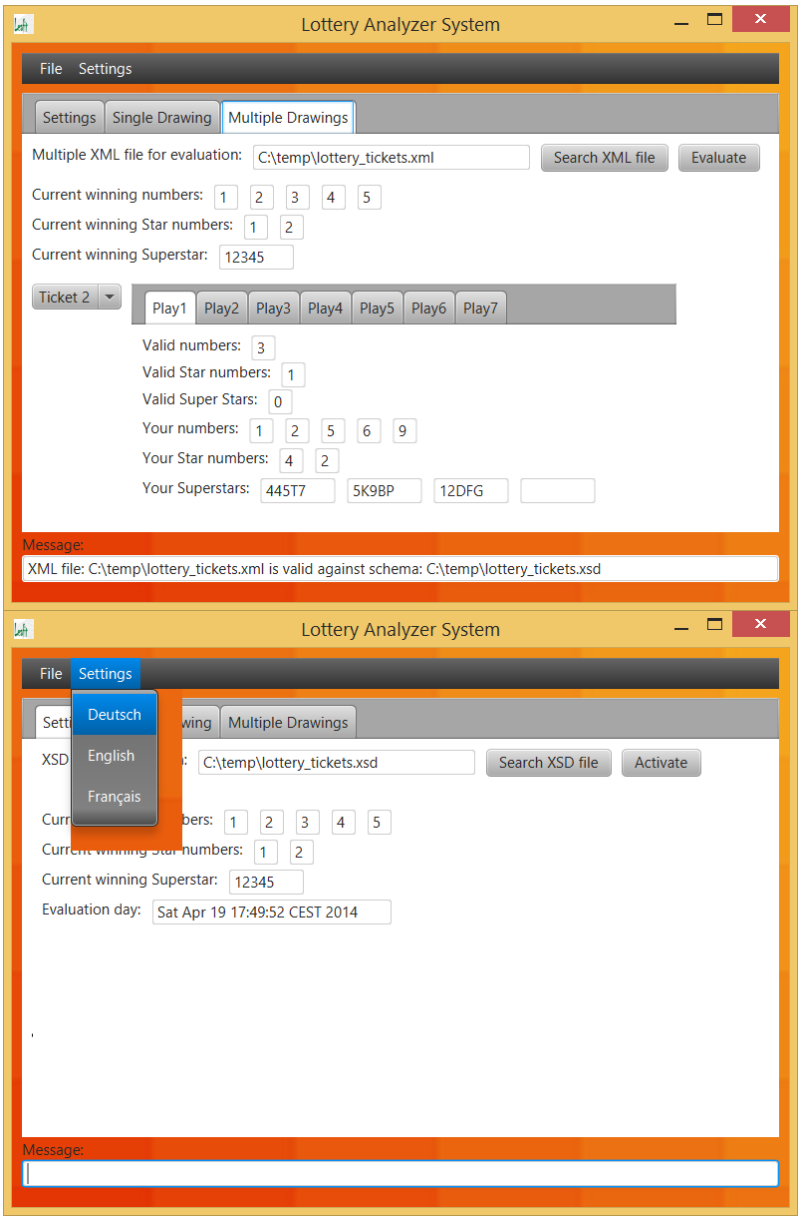
- Entwerfen Sie ein **XML Schema (XSD)** welches die **obigen Rahmenbedingungen** abdeckt und es ermöglicht „*Lotto-XML-Files*“ zu validieren. Erstellen Sie das XSD so, dass es sowohl für die Validierung von XML Files für **einzelne** Lottoscheine und für die Validierung von XML Files, die eine **beliebige Anzahl** von Lottoschein-Informationen enthalten, verwendet werden kann.
- Schreiben Sie eine **JavaFX Anwendung**, die mindestens folgende Funktionalitäten besitzt:
    - Benutzerfreundliches GUI (multilingual für English und Deutsch).
    - Einlesen des Schemas (XSD File).
    - Einlesen und Analysieren eines dem Schema entsprechenden XML File (für beide Möglichkeiten: 1 Lottoschein, mehrere Lottoscheine).
    - Validierung der XML Files.
    - Auswertungen der jeweiligen eingelesenen XML Informationen.
    - Die Auswertung soll pro Lottospiel eine Auswertung anzeigen – entsprechend obiger Figur.

Angaben zur Implementierung:

- Nutzen Sie JAXB für die Auswertungen der XML Files.
- Nutzen Sie CSS Files für das grafische Layout der JavaFX Anwendung.

Folgende Figuren zeigen wie die Anwendung aussehen kann (Sie sind aber aufgefordert, kreativ zu sein und eine bessere Lösung zu erstellen).


Lottery Analyzer System
File Settings
Settings Single Drawing Multiple Drawings
XSD file for validation: C:\temp\lottery_tickets.xsd
Search XSD file
Activate
Current winning numbers: 1 2 3 4 5
Current winning Star numbers: 1 2
Current winning Superstar: 12345
Evaluation day: Sat Apr 19 17:49:52 CEST 2014
Message:
All OK...
Lottery Analyzer System
File Settings
Settings Single Drawing Multiple Drawings
Single XML file for evaluation: C:\temp\lottery_ticket.xml
Search XML file
Evaluate
Valid numbers: 5
Valid Star numbers: 1
Valid Super Stars: 0
Current winning numbers: 1 2 3 4 5
Current winning Star numbers: 1 2
Current winning Superstar: 12345
Your numbers: 1 2 3 4 5
Your Star numbers: 1 8
Your Superstars: 22345 33457
Message:
XML file: C:\temp\lottery_ticket.xml is valid against schema: C:\temp\lottery_tickets.xsd


Lottery Analyzer System
File Settings
Settings Single Drawing Multiple Drawings
Multiple XML file for evaluation: C:\temp\lottery_tickets.xml
Search XML file
Evaluate
Current winning numbers: 1 2 3 4 5
Current winning Star numbers: 1 2
Current winning Superstar: 12345
Ticket 2
Play1 Play2 Play3 Play4 Play5 Play6 Play7
Valid numbers: 3
Valid Star numbers: 1
Valid Super Stars: 0
Your numbers: 1 2 5 6 9
Your Star numbers: 4 2
Your Superstars: 445T7 5K9BP 12DFG
Message:
XML file: C:\temp\lottery_tickets.xml is valid against schema: C:\temp\lottery_tickets.xsd
Lottery Analyzer System
File Settings
Deutsch
English
Français
Multiple Drawings
C:\temp\lottery_tickets.xsd
Search XSD file
Activate
Current winning Superstar: 12345
Evaluation day: Sat Apr 19 17:49:52 CEST 2014
Message:

2 Escaping a labyrinth

Write a **graphical application** (Swing or JavaFX) that displays a labyrinth (see the example below) and that shows the way out starting from an entered position.

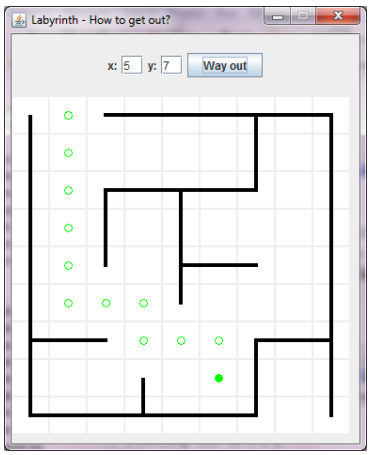
The labyrinth should be **initialized from a text or xml file with an appropriate format**, e.g. like the given example. The numbers in the first line indicate the dimensions of the labyrinth. The walls of the labyrinth are indicated by asterisks (*). Use a class `Tile` to represent the different parts of the labyrinth (wall or empty), and a class `LabyrinthModel` to define the structure of the whole labyrinth.

`Maze1.txt`

```
9,9
* * * * * * * * *
*       *       *
* * * * * * * * *
*   *   *       *
* * * * * * * * *
*   *     *     *
* * * * * * * * *
*       *       *
* * *       * * *
*   *   *   *   *
* * * * * * * * *
```

In your application, provide 2 input fields to enter the row and column of the start position in the labyrinth. When the button “Way out” is pressed, compute the escape way from the labyrinth and display it.

Implement a **recursive method** in the class `LabyrinthModel` to find the way out. Use the following recursive approach to check whether you can escape from the labyrinth: If you are at an exit (an exit is an empty tile at the border of the labyrinth), return true. Recursively check whether you can escape from one of the empty neighboring tiles without visiting the current location (mark the tiles that you have already checked).



3 Thread Execution

3.1 Version 1

Implement an application containing *n* threads, that calculate a list of prime numbers *(2, 3, 5, 7, 11, …)*. Proceed as follows:

- In the `main` method, start these *n* threads and let them calculate prime numbers as long as they are not interrupted.
- Also start a calculation in the main method, which calculates the list of prime numbers until the maximum number *m*.
- When the calculation of prime numbers *(2, 3, …, m)* is completed in the `main` method, interrupt all the other running threads.
- When all threads (except the `main` one) have been terminated, display for each thread the highest prime number that was calculated.

Please note:

- Each thread should sleep randomly between 0 and 1 s between each calculation step.
- Name every thread and distribute the output to the console conveniently in order to observer which thread is running or waiting (thread state).
- To check, if a number is a prime number, use the following method:

```
public static boolean isPrime(int no) {
    if (no < 2) return false;
    for (int i = 2; i < no; i++)
      if (no % i == 0) return false;
    return true;
}
```

3.2 Version 2

Implement a second version of an application that calculates prime numbers. Proceed as follows:

- In the `main` thread, calculate a list of prime numbers until the maximum value *m* and store these numbers in an array list.
- Recalculate the same prime numbers, but divide the calculation equally between m parallel threads. E.g., when you decide to calculate prime numbers until 1000 with 5 threads, the first thread should calculate prime numbers in the interval 1..200, the second in the interval 201..400, etc.

- Each thread must store the prime number it calculates in a **local** array list and when it terminates, it should return this list to the caller (use `Callable` object).
- At program end, all partial lists (one for each thread) must be stored together in an array list which must have the same content as the array list built in the *main* thread. Check this!

Compare the time needed to execute the calculations with one thread and with n parallel threads (Hint: Introduce a very short delay (1 ms) between each call of the *isPrime* method).