# ASB-Aufgabe zu FOPT 1 (Pflicht)

In dieser Aufgabe sollen Sie die Funktionsweise einer Ampel und deren Nutzung nachahmen.

a) Schreiben Sie zunächst eine Schnittstelle Ampel mit zwei parameterlosen void-Methoden schalteRot und schalteGruen zum Setzen der Ampel auf rot bzw. grün (die Zwischenphase gelb spielt keine Rolle – Sie können von diesem Zustand abstrahieren)! Eine dritte parameterlose void-Methode mit dem Namen passieren soll das Vorbeifahren eines Fahrzeugs an dieser Ampel nachbilden. Mit einer vierten parameterlosen int-Methode namens wartendeFahrzeuge sollen Sie abfragen können, wie viele Autos momentan an der Ampel stehen.

b) Geben Sie jetzt eine erste Implementierung der Schnittstelle Ampel namens ItalienischeAmpel an! Die Methode passieren soll dabei folgendermaßen funktionieren: Ist die Ampel beim Aufruf der Methode rot, so wird der aufrufende Thread angehalten, und zwar so lange, bis die Ampel grün wird. Ist die Ampel dagegen grün, so kann der Thread sofort aus der Methode zurückkehren, denn es muss nicht auf einen Zustandswechsel der Ampel gewartet werden. Sie müssen zunächst nicht wie in der Realität nachahmen, dass beim Grünwerden der Ampel die Autos der Reihe nach losfahren. Stellen Sie sich der Einfachheit halber eine Ampel vor, bei der sich alle wartenden Autos nebeneinander aufstellen (wie in Italien)! Sobald die Ampel grün wird, hat jedes Auto unabhängig von allen anderen Autos die Möglichkeit loszufahren. Verwenden Sie wait, notify und notifyAll nur an den unbedingt nötigen Stellen!

c) Geben Sie nun eine zweite Implementierung der Schnittstelle Ampel mit dem Namen DeutscheAmpel an! Bei dieser Implementierung sollen sich die Autos nicht mehr nebeneinander aufstellen, sondern hintereinander! Das heißt, ein Auto kann erst fahren, wenn die Ampel grün ist und alle vor ihm stehenden Autos an der Ampel vorbeigefahren sind. Hier ist also die Reihenfolge, in der die Autos an der Ampel stehen, zu beachten.

d) Schreiben Sie nun eine Klasse Auto (abgeleitet aus Thread)! Im Konstruktor wird eine Referenz auf ein Feld von Ampeln übergeben, wobei diese Referenz in einem entsprechenden Attribut der Klasse Auto gespeichert wird. In der run-Methode werden alle Ampeln des Feldes passiert, und zwar in einer Endlosschleife (d.h. nach dem Passieren der letzten Ampel des Feldes wird wieder die erste Ampel im Feld passiert). Verwenden Sie als Typ die Schnittstelle Ampel, so dass nicht festgelegt wird, um welche Art von Ampel es sich wirklich handelt!

e) Schreiben Sie eine weitere Thread-Klasse zur Steuerung der Ampeln. Der Konstruktor erhält wiederum eine Referenz auf ein Feld von Ampeln. Verwenden Sie auch hier wieder als Typ die Schnittstelle Ampel, so dass nicht festgelegt wird, um welche Art von Ampel es sich wirklich handelt!

f) Probieren Sie Ihre Threads mit italienischen und deutschen Ampeln aus!

**Hinweise für ASB:**

- Alle Klassen müssen sich im Package pp.ampel befinden
- Das Interface Ampel und alle darin deklarierten Methoden müssen öffentlich sein.
- Die Klassen ItalienischeAmpel und DeutscheAmpel müssen beide einen öffentlichen parameterlosen Konstruktor haben.

# ASB-Aufgabe zu FOPT 2 (Pflicht)

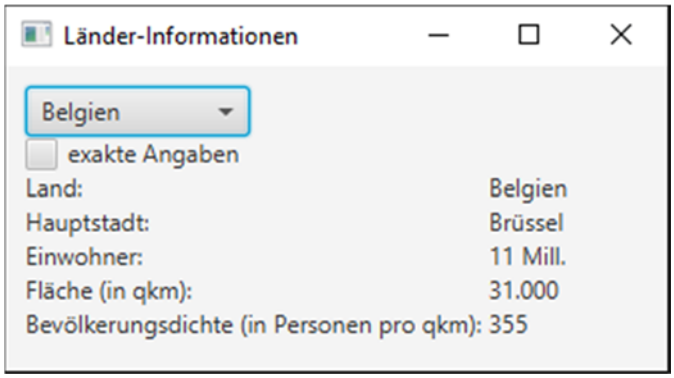
Gegeben sei die wohlbekannte Datenstruktur eines Kellers (Stack) mit den Methoden push und pop. Im Folgenden sollen diese Methoden so geändert werden, dass die Methode push (Ablegen eines Elements im Keller) wie allgemein üblich ohne besondere Einschränkungen ausgeführt werden kann (d.h. die Größe des Kellers soll nicht durch Ihr Programm beschränkt werden). Beim Aufruf der Methode pop (Entfernen des obersten Elements des Kellers und Rückgabe des entfernten Elements als Rückgabewert) soll allerdings bei leerem Keller so lange gewartet werden, bis ein Element zurückgegeben werden kann. Das heißt: Es kann nicht vorkommen, dass die Methode pop kein Element zurückgibt, da der Keller leer ist. Die Elemente, die im Keller abgelegt werden, sollen Objekte jeder beliebigen Klasse sein können. Deshalb soll als Datentyp für die abzulegenden Elemente die Klasse Object verwendet werden. Der Keller soll durch ein Feld (in diesem Fall ein Object-Feld) oder eine ArrayList realisiert werden.
Implementieren Sie den Keller als Klasse SynchStack! Benutzen Sie zur Synchronisation ausschließlich Semaphore (d.h. Sie sollen die Klasse Semaphore aus dem Lehrbuch verwenden, Sie dürfen aber sonst in Ihrem Programm an keiner anderen Stelle synchronized, wait, notify und notifyAll einsetzen)! Die Synchronisation mit Semaphoren soll sich ausschließlich in der Kellerklasse SynchStack befinden (d.h. Benutzer dieser Klasse sollen mit der Synchronisation nichts zu tun haben).
Bitte machen Sie sich zur Lösung dieser Aufgabe klar, dass Sie nicht nur wait und notify bzw. notifyAll mit Hilfe eines Semaphors realisieren müssen, sondern auch synchronized! Das heißt: Es muss durch Ihre Implementierung verhindert werden, dass zwei Threads gleichzeitig auf die Attribute des Kellers (Object-Feld bzw. ArrayList) zugreifen.

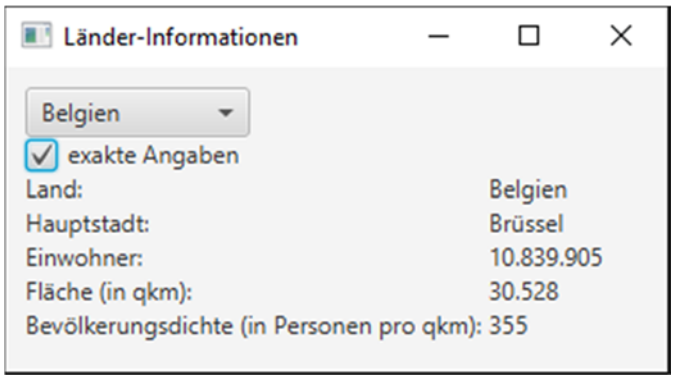
**Hinweise für ASB:**

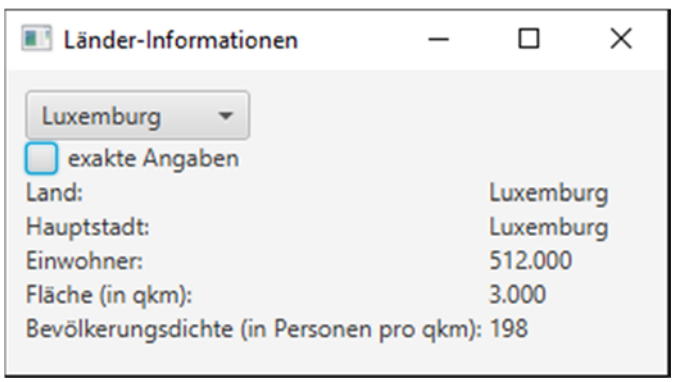
- Die Klasse SynchStack muss sich im Package pp.synchstacksem befinden.
- Die Klasse SynchStack muss einen öffentlichen parameterlosen Konstruktor haben.
- Die im Lehrbuch benutzte Klasse Semaphore muss in der hochgeladenen ZIP-Datei enthalten sein. Sie soll sich ebenfalls im Package pp.synchstacksem befinden.

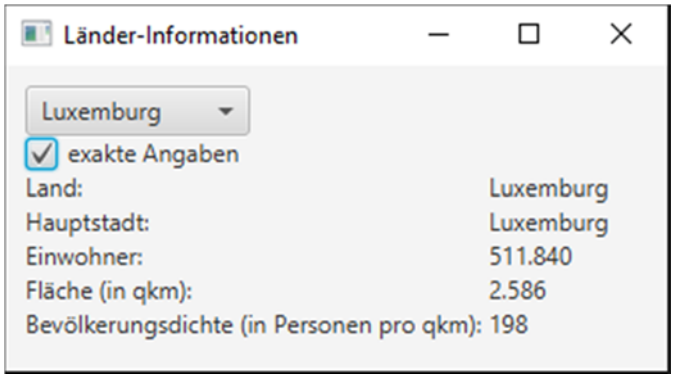
# ASB-Aufgabe zu FOPT 3 (Pflicht)

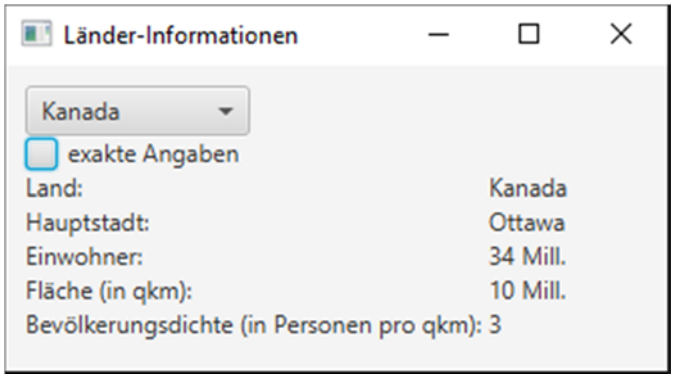
In dieser Aufgabe soll gezeigt werden, dass es sinnvoll sein kann, zu einer ComboBox nicht Strings, sondern Objekte eines anderen Typs hinzuzufügen. Im hier zu entwickelnden Programm werden Objekte der Klasse Country zu einer ComboBox hinzugefügt. Die Klasse Country beschreibt ein Land wie etwa Deutschland, Frankreich, Luxemburg usw. durch den Ländernamen, den Namen der Hauptstadt, die Anzahl der Einwohner sowie die Fläche (in qkm).
Wird ein Land aus der ComboBox ausgewählt, so werden die Länderinformationen in mehreren Labels dargestellt. Im Fenster wird zusätzlich die aus der Einwohnerzahl und der Fläche errechnete Bevölkerungsdichte angezeigt. Mit Hilfe einer CheckBox kann festgelegt werden, ob für die Einwohnerzahl und die Fläche genau die Werte angezeigt werden, die im entsprechenden Land-Objekt vorhanden sind, oder ob übersichtlicher auf Tausend bzw. auf eine Million gerundete Werte zu sehen sind.

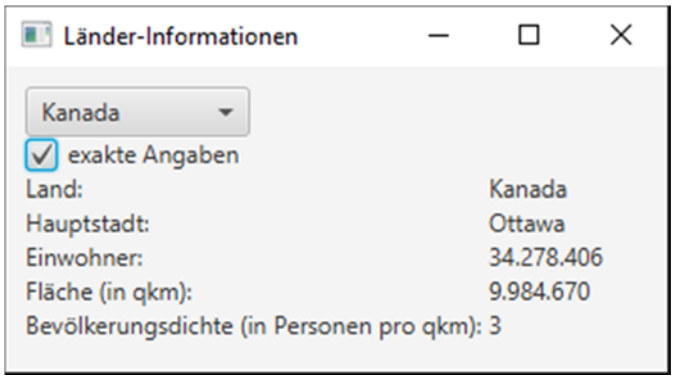





a) Ergänzen Sie die öffentliche Klasse Land so, dass die Anzeige der JComboBox so ist wie in den Abbildungen auf der vorigen Seite! Ein Attribut für die Bevölkerungsdichte ist - wie unten zu sehen - nicht unbedingt notwendig, da diese Angabe jedes Mal errechnet werden kann. Sollten Sie ein solches Attribut mit einer entsprechenden Get-Methode aber hinzufügen wollen, um sich die Neuberechnung der Dichte bei jeder Anzeige zu sparen, so können Sie das gerne machen.

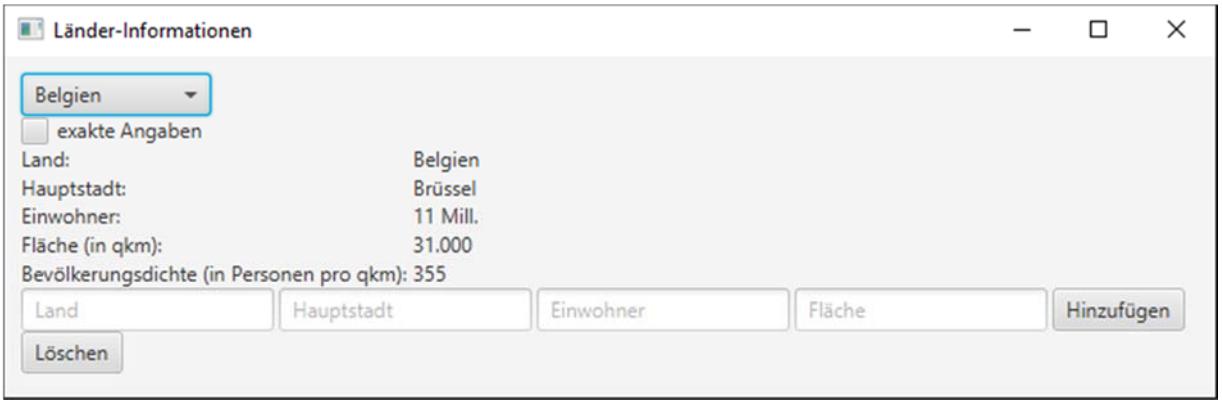
```
public class Country
{
    private String name;
    private String capital;
    private long people;
    private long area;

    public Country(String name, String capital,
                   long people, long area)
    {
        this.name = name;
        this.capital = capital;
        this.people = people;
```

```
            this.area = area;
        }

        public String getName()
        {
            return name;
        }

        public String getCapital()
        {
            return capital;
        }

        public long getPeople()
        {
            return people;
        }

        public long getArea()
        {
            return area;
        }

        //Methode zu ergänzen (welche und wie?)
->
    }
```

b) In dieser Teilaufgabe geht es um den Aufbau der Oberfläche. Außer der ComboBox soll es 10 Labels geben, und zwar 5 für die Beschriftungen „Land:“, „Hauptstadt:“, „Einwohner:“ usw. (auf der linken Seite des Fensters) und 5 für die dazugehörigen Werte wie „Belgien“, „Brüssel“, „11 Mill.“ usw. (auf der rechten Seite des Fensters). Achten Sie darauf, dass nach dem Starten des Programms bereits ein Land ausgewählt ist (z.B. das erste in der ComboBox)!

c) Programmieren Sie nun die Ereignisbehandlung! Sie müssen auf das An- und Abwählen der CheckBox sowie auf das Auswählen eines Landes in der ComboBox reagieren. Die Ausgabe für die Anzahl der Einwohner und für die Fläche werden auf Millionen (z.B. „80 Mill.“) oder auf Tausend (z.B. „378.000“) gerundet. Schreiben Sie zu diesem Zweck eine entsprechende Formatierungsmethode.

d) Erweitern Sie Ihr Programm nun so, dass zusätzliche Länder hinzugefügt sowie das aktuell angezeigte Land gelöscht werden kann:



Achten Sie beim Hinzufügen darauf, dass nur gültige Angaben (nicht leere Strings für Land und Hauptstadt sowie gültige Zahlenangaben für Einwohner und Fläche) übernommen werden! Nach dem Drücken des Buttons „Hinzufügen“ sollen automatisch alle Texte in den Eingabefeldern gelöscht werden. Wenn Sie ein Land bei leerer Auswahlliste hinzufügen, dann soll nach dem Hinzufügen automatisch das neu hinzugefügte Land angezeigt werden.

Wenn beim Löschen nur noch ein Element vorhanden ist, dann soll die Anzeige in den Labels auf der rechten Seite verschwinden:

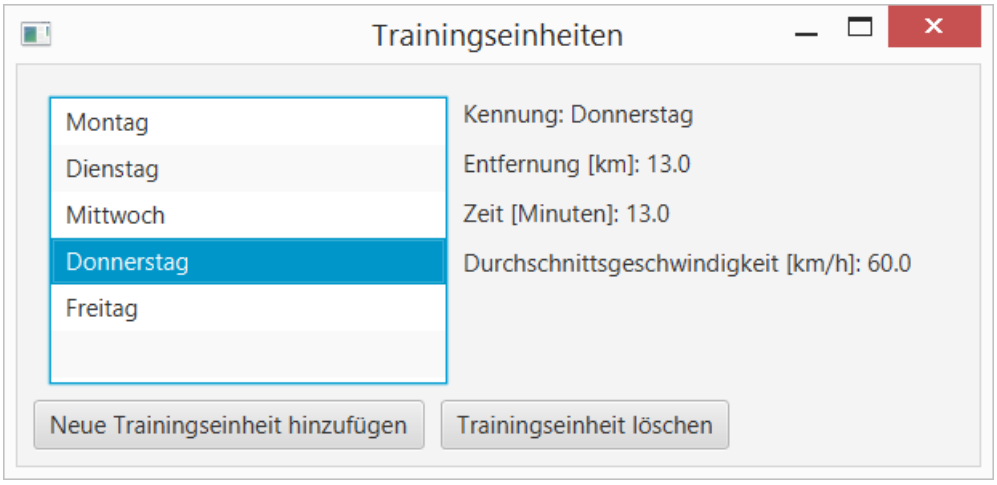

Wenn das erste Element in der Liste gelöscht wird, kann es zu einer Ausnahme kommen. Achten Sie darauf, dass es nicht zu dieser Ausnahme kommt!

**Hinweise für ASB:**

- Ihre Dateien (u.a. die Klasse Country) sollen sich im Package gui.country.combo befinden.
- Ihre aus Application abgeleitete Klasse soll CountryInfo heißen.
- Die ComboBox muss die ID „countrySelector“ tragen und einige Beispielländer zur Auswahl anbieten.
- Die CheckBox zur Festlegung, ob genaue Werte angezeigt werden sollen, muss die ID „exactValues“ haben.
- Die Labels, die die Länderinformationen anzeigen, müssen die IDs „countryName“, „capital“, „population“, „area“ und „density“ besitzen.
- Die Eingabefelder zum Hinzufügen neuer Länder müssen die IDs „countryField“, „capitalField“, „populationField“ und „areaField“ besitzen.
- Der Hinzufügen-Button muss die ID „add“, der Löschen-Button die ID „delete“ haben.
- Die Beschriftungen von Elementen, die sich während der gesamten Programmausführung nicht ändern, d.h die nicht von der Auswahl eines bestimmten Landes abhängen, müssen mit den Beschriftungen in den Abbildungen übereinstimmen.
- Verwenden Sie in Ihrem Programm nicht den String „Länder“, „Hinzufügen“ oder „Löschen“, sondern „L\u00e4nder“, „Hinzuf\u00fcgen“ und „L\u00f6schen“. Für die Umlaute kann es je nach Einstellung unterschiedliche Codierungen geben. Dadurch könnte es sein, dass der Button mit der Beschriftung „Hinzufügen“ von den ASB-Tests nicht gefunden wird, obwohl er da ist.

# ASB-Aufgabe zu FOPT 4 (Pflicht)

Überblick: In dieser Aufgabe geht es um ein Programm, mit dem man seine Trainingseinheiten bestehend aus der Trainingsstreckenlänge und der dafür benötigten Zeit unter einer frei wählbaren Kennung (Datum, Uhrzeit, Wochentag, Sportart wie Jogging, Radfahren, Rudern usw.) eintragen, anzeigen und löschen kann. Die Oberfläche zeigt eine Übersicht aller Trainingseinheiten in einer ListView. Wird eine davon selektiert, dann sieht man rechts neben der Liste die Streckenlänge (in km), die Zeit (in Minuten) und die sich daraus ergebende Durchschnittsgeschwindigkeit (in km/h) für die selektierte Trainingseinheit. Weiterhin gibt es Buttons zum Hinzufügen neuer Trainingseinheiten sowie zum Löschen vorhandener Trainingseinheiten:




Das Programm soll nach dem MVP-Prinzip realisiert werden. Im Einzelnen sollen folgende Vorgaben eingehalten werden:

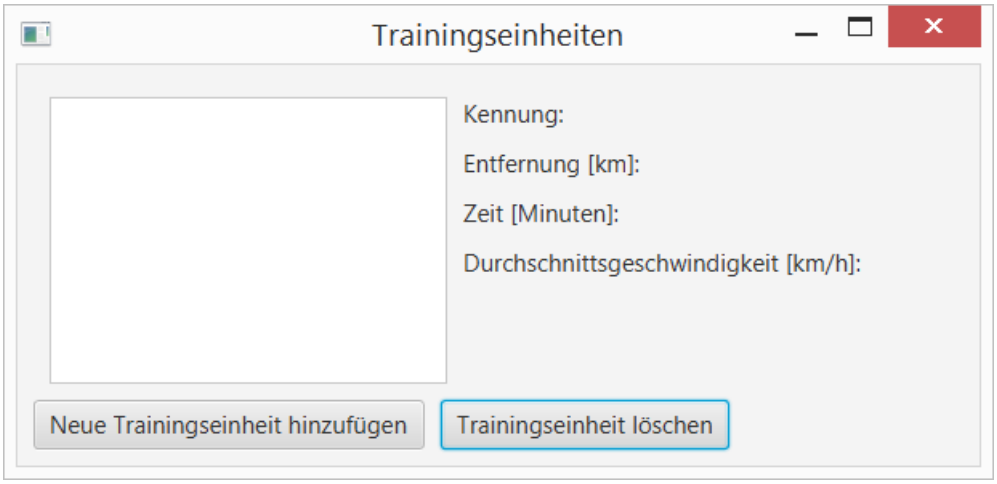
a) In dieser ersten Teilaufgabe soll es um das Modell gehen. Das Modell besteht aus 2 Klassen:

- Die Klasse *TrainingUnit* beschreibt eine einzige Trainingseinheit. Sie hat Attribute zum Speichern der Kennung (beliebiger String ohne Vorgaben), der Entfernung (float) und der Zeit (float). Ferner besitzt sie einen public-Konstruktor, mit dem alle Attribute durch Parameter (in der hier angegebenen Reihenfolge) initialisiert werden. Ein Objekt der Klasse TrainingUnit ist nach der Initialisierung nicht mehr änderbar. Alle weiteren Methoden sind öffentliche und lesende Methoden. Sie haben die Namen *getMarker*, *getDistance*, *getTime* und *getMeanSpeed*. Außerdem soll eine weitere Methode für die Darstellung in der ListView geeignet überschrieben werden. Der Code dieser Klasse wird Ihnen vollständig zur Verfügung gestellt.
- Die Klasse *Model* ist die Hauptklasse der Modellkomponente. Sie beinhaltet eine beliebige Anzahl von TrainingUnit-Objekten. Die Klasse *Model* soll einen öffentlichen, parameterlosen Konstruktor besitzen. Außerdem soll sie eine öffentliche void-Methode namens *addTrainingUnit* mit einem TrainingUnit-Parameter besitzen, mit der eine neue Trainingseinheit dem Modell hinzugefügt werden kann. Die Kennungen für die Trainingseinheiten müssen eindeutig sein. Wenn die Kennung schon vorhanden ist, dann soll diese Methode eine IllegalArgumentException werfen. Weitere Methoden der Klasse *Model* sollen sein: eine öffentliche Methode *removeTrainingUnit* zum Löschen eines Eintrags (void, Parameter des Typs String zur Angabe der

Kennung der zu löschenden Einheit), *getTrainingUnit* zum Auslesen eines Eintrags (öffentlich, Rückgabetyp *TrainingUnit*, Parameter des Typs String zur Angabe der Kennung der gesuchten Einheit) sowie *getAllMarkers* zum Auslesen der Kennungen aller momentan vorhandenen Trainingseinheiten (öffentlich, Rückgabetyp String[], keine Parameter).

b) Implementieren Sie nun noch View, Presenter und Main so, dass das Auswählen eines Eintrags aus der ListView und die Anzeige der Details zur ausgewählten Trainingseinheit möglich ist! Belegen Sie in Main Ihr Modell mit einigen Trainingseinheiten, damit Sie Ihre erste lauffähige Version ausprobieren können!

c) Erweitern Sie nun Ihre erste lauffähige Anwendung so, dass der aktuell selektierte Eintrag gelöscht werden kann! Achten Sie dabei darauf, dass für den Fall, dass der letzte Eintrag gelöscht wurde, die Detailansicht keine Anzeige mehr enthält:




d) Realisieren Sie nun abschließend die Möglichkeit, über einen modalen Dialog neue Trainingseinheiten hinzufügen zu können! Der Dialog soll erscheinen, wenn Sie einen entsprechenden Button („Neue Trainingseinheiten hinzufügen“ auf der Abbildung der vorigen Seite) im Fenster drücken. Der Dialog könnte in etwa so aussehen:


Schließt man diesen Dialog nicht über „Hinzufügen“, soll außer dem Verschwinden nichts Weiteres passieren. Klickt man auf „Hinzufügen“, dann verschwindet der Dialog, eine neue Trainingseinheit wird hinzugefügt und erscheint dann auch sofort in der ListView, falls eine nicht leere, noch nicht verwendete Kennung eingegeben wurde und die Strings in den beiden restlichen Feldern sich zu einer float-Zahl parsen lassen (Float.parseFloat, wirft Ausnahme NumberFormatException, falls das Parsen

nicht gelingt). Andernfalls wird eine entsprechende Fehlermeldung am unteren Rand des Dialogs angezeigt:




Außerdem muss die Kennung eindeutig sein. Wenn versucht wird, eine TrainingUnit dem Modell hinzuzufügen mit einer Kennung, die es schon gibt, dann wird nach Aufgabenteil a eine Ausnahme geworfen. Diese Ausnahme soll zu der Fehlermeldung „Kennung: existiert schon“ am unteren Rand des Dialogs führen.

Den Dialog sollen Sie in einer eigenen Klasse namens EditorDialog realisieren, die Sie aus Stage ableiten sollten. Um den Dialog modal zu setzen, wenden Sie auf Ihre aus Stage abgeleitete Klasse die Methode initModality mit dem Parameter Modality.APPLICATION_MODAL an! Zum Anzeigen des Dialogs wenden Sie bitte die Methode showAndWait an! Diese Methode kehrt erst zurück, wenn der Dialog geschlossen wurde. Zum Schließen über den Button „Abbrechen“ wenden Sie bitte die Methode close (parameterlos) auf Ihr Dialog-Objekt an! Das Schließen des Dialogs über das Kreuz oben rechts ist quasi schon vorprogrammiert.

Das Anzeigen des Dialogs kann allein von der View durchgeführt werden. Es muss dazu nicht der Presenter beauftragt werden. Wenn der Dialog über „Hinzufügen“ geschlossen wird, dann soll

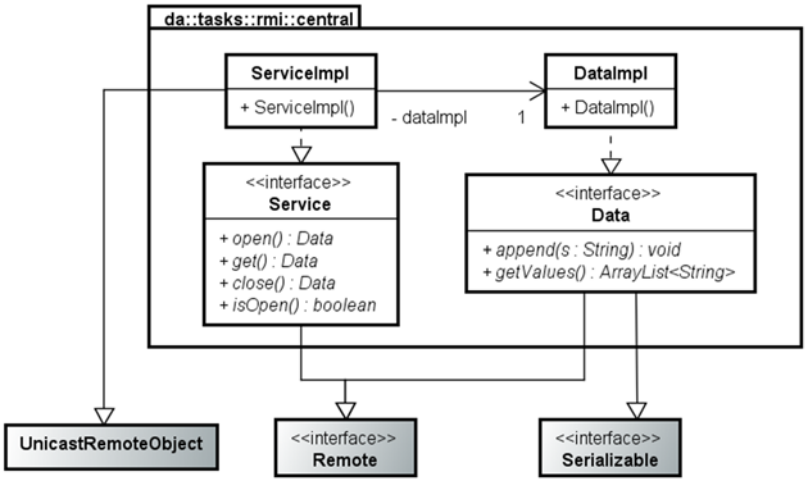
showDialog eine Trainingseinheit zurückliefern, die dann von der View an den Presenter übergeben wird.

Noch ein Hinweis: Sie können und sollen die Aufgabe vollständig ohne static lösen.

**Hinweise für ASB:**

- Alle Klassen müssen sich im Package gui.mvp.training befinden und öffentlich (public) sein.
- Die ListView muss die ID „overviewList“ haben.
- Die Labels zur Anzeige der Details einer Traininseinheit müssen die ID’s „markerLabel“, „distance-Label“, „timeLabel“ und „meanSpeedLabel“ haben.
- Die Textfelder und das Label zur Anzeige eventueller Fehler müssen die ID’s „markerTF“, „distanceTF“, „timeTF“ und „errMsgLabel“ haben.
- Die Beschriftungen der Buttons sowie die Fehlermeldungen müssen genau mit denen der Aufgaben-stellung übereinstimmen.
- Verwenden Sie zur einheitlichen Darstellung von Umlauten Unicode-Zeichen! Bei unterschiedlichen Codierungen kann es sonst vorkommen, dass beispielsweise ein Button von den ASB-Tests nicht gefunden wird, obwohl er vorhanden ist. Verwenden Sie also zum Beispiel nicht den String „Neue Trainingseinheit hinzufügen“, sondern „Neue Trainingseinheit hinzuf\u00FCgen“.

# ASB-Aufgabe zu FOPT 5 (Pflicht): Zentralisierte Sammlung von Daten per RMI




Gegenstand dieser Aufgabe ist die zentralisierte Sammlung von Daten, hier simuliert durch Zeichenketten. Dazu ist von einem Server eine Instanz des Typs `Service` per RMI öffentlich zur Verfügung zu stellen (siehe Abbildung oben). Clients erhalten über sie Zugriff auf das eigentliche Datenhaltungsobjekt des Typs `Data`. Die Methode `get()` gibt abhängig vom Zustand der Datensammlung eine Referenz auf das Objekt beziehungsweise eine Wertkopie desselben zurück.

Im Anfangszustand *Offen* erfolgt Referenzrückgabe. Durch Aufruf der Methode `close()` durch einen beliebigen Client wird die Datensammlung *geschlossen*. Clients erhalten nunmehr nur noch eine Wertkopie des Objekts. Erst mit dem neuerlichen *Öffnen* der Datensammlung über `open()` wird der Anfangszustand wiederhergestellt.

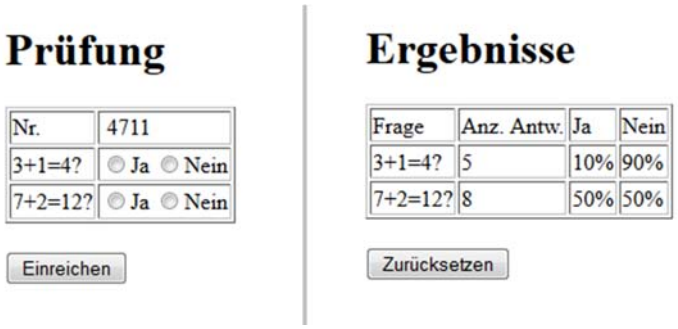
Die Methoden `open()` und `close()` geben das Datenobjekt auf die Art und Weise zurück, die dem Zustand entspricht, der durch sie erreicht wird.

Die Schnittstelle `Data` schreibt die Methoden `append(...)` und `getValues()` vor. Erstere soll den Clients dazu dienen, der Datensammlung neue Einträge in Form von Zeichenketten hinzuzufügen, letztere soll diese Einträge zurückgeben.

**Hinweise für ASB:**

- Legen Sie Ihre Klassen und Schnittstellen im Paket `da.tasks.rmi.central` ab.
- Alle Klassen und Schnittstellen müssen öffentlich sein.
- Ihre Lösung muss die in der obigen Abbildung dargestellten Klassen und Schnittstellen mit den angegebenen Methoden und Attributen enthalten. Ausgenommen sind die grau hinterlegten Klassen, die Teil des Java-Funktionsumfangs sind.
- Die dargestellten Schnittstellen sind von `Remote` abzuleiten, die Klassen von `UnicastRemoteObject`.

# ASB-Aufgabe zu FOPT 6 (Pflicht): Prüfungssystem auf Basis von Servlets




Thema dieser Aufgabe ist ein Prüfungssystem, das mit Hilfe von Servlets realisiert werden soll. Es soll zwei Ansichten enthalten, die optisch der oben abgebildeten Skizze entsprechen.

Den Prüfungsteilnehmern steht eine Prüfungsseite zur Verfügung, auf der eine vom System zu vergebende Prüfungsnummer sowie zwei Prüfungsfragen zu finden sind, die einfache Ja/Nein-Probleme sind. Antworten können mit einem entsprechenden Knopf eingereicht werden.

Für den Prüfer ist eine Ergebnisseite vorzusehen, auf der zu jeder Frage die Anzahl der eingegangenen Antworten und die Prozentsätze der Ja- beziehungsweise der Nein-Antworten dargestellt sind. Zusätzlich soll er die Möglichkeit haben, die Prüfung zurückzusetzen, also alle eingegangenen Antworten zu löschen.

### Teilaufgabe a

Die erste Teilaufgabe besteht in der Realisierung der Prüfungsseite in Form der Klasse `ExamServlet`.

Die Prüfungsseite soll genau ein Formular enthalten, welches seinerseits zunächst einmal die Prüfungsnummer enthält. Diese ist beim erstmaligen Aufruf des Servlets neu zu bestimmen und soll dazu dienen, die Prüfung beziehungsweise den Prüfungsteilnehmer zu identifizieren. Es darf daher nicht möglich sein, sie zu ändern.

Insgesamt soll das Formular Folgendes enthalten:

- Ein `input`-Element mit dem Namen `id` zur Aufnahme der Prüfungsnummer. Wählen Sie für dieses Element einen passenden Typ.
- Zwei `input`-Elemente des Typs `radio`, beide mit dem Namen `q1` und den Werten `y` (Ja) beziehungsweise `n` (Nein) zur Aufnahme der Antwort auf Frage 1.
- Zwei entsprechende Elemente mit dem Namen `q2` für Frage 2.
- Einen Knopf mit der Bezeichnung "Einreichen".

Auf das Betätigen des Knopfes hin soll die Nummer zusammen mit den angegebenen Antworten per HTTP-GET an das `ExamServlet` weitergegeben werden.

Diese Daten sind sodann als Attribute im `ServletContext` abzulegen. Bezüglich der Attributnamen soll dabei dem Schema `exam-q[n]-[id]` gefolgt werden, wobei `[n]` die Nummer der Frage und `[id]` die Nummer der Prüfung ist. Als Attributwerte sind Instanzen der Klasse `Boolean` zu verwenden. Unbeantwortete Fragen sind dort nicht abzulegen.

Zur Verdeutlichung sei beispielhaft angenommen, dass durch zwei Prüfungsteilnehmer Antworten eingereicht wurden. Der Teilnehmer mit der Nummer 1 hat dabei beide Fragen mit "Ja" beantwortet, der Teilnehmer Nummer 2 hat hingegen nur die erste Frage mit "Nein" beantwortet. In diesem Fall müssen sich folgende Attribute im `ServletContext` befinden:

- `exam-q1-1 = Boolean.TRUE`
- `exam-q2-1 = Boolean.TRUE`
- `exam-q1-2 = Boolean.FALSE`

**Teilaufgabe b**

1) In einem ersten Schritt sollen die von einem Teilnehmer eingereichten Antworten inklusive seiner Prüfungsnummer in der Session des jeweiligen Benutzers gespeichert werden. Bei dem nächsten Aufruf des `ExamServlet` sollen sie dann innerhalb des Formulars vorausgewählt werden. Prüfen Sie, was passiert, wenn der Benutzer vor ebendiesem nächsten Aufruf seinen Browser neu startet.

2) Statt der Session sollen nun direkt Cookies verwendet werden. Hierbei ist sicherzustellen, dass die Daten auch dann noch zur Verfügung stehen, wenn der Browser zwischenzeitlich geschlossen wurde.

Speichern Sie die jeweilige Prüfungsnummer in einem Cookie mit dem Namen `exam-id`. Die eingereichten Antworten sind, sofern vorhanden, in Cookies mit den Namen `exam-q1` beziehungsweise `exam-q2` abzulegen. Speichern Sie hier je nach Antwort die Zeichenkette `true` oder `false`. Zu unbeantworteten Fragen müssen keine Cookies angelegt werden.

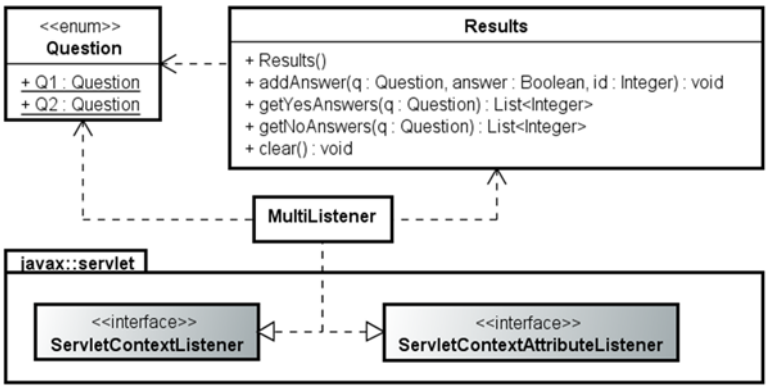
**Teilaufgabe c**

Zur Vorbereitung der Umsetzung der Ergebnisseite ist zunächst die Klasse `Results` zu implementieren. Diese soll dazu dienen die Antworten der Prüfungsteilnehmer zu sammeln. Hierzu soll es möglich sein, pro Prüfungsfrage die Prüfungsnummern derjenigen Personen zu speichern, die die jeweilige Frage mit "Ja" beziehungsweise mit "Nein" beantwortet haben.

Es ist dafür zu sorgen, dass beim Start des verwendeten Servers eine Instanz der Klasse `Results` im `ServletContext` gespeichert wird, die dann unter dem Attributnamen `results` zentral zur Verfügung steht. Verwenden Sie hierzu einen `Servlet-ContextListener`.

Weiterhin ist ein `ServletAttributeContextListener` zu implementieren, der beim Ablegen von Antworten auf Prüfungsfragen im `ServletContext` in Erscheinung tritt. Er

soll dann die Inhalte des zentralen `Results`-Objekts gemäß der jeweils neuen Antwort aktualisieren.




Implementieren Sie die beiden oben genannten Listener-Schnittstellen gemeinsam in der Klasse `MultiListener`. Die Klasse `Results` soll dem obigen Klassendiagramm entsprechen. Die Funktionen der einzelnen dort aufgeführten Methoden sollten selbsterklärend sein. Verwenden Sie außerdem einen Aufzählungstyp `Question` mit dargestellten Literalen zur Identifikation der beiden Prüfungsfragen.

**Teilaufgabe d**

Nun ist die Klasse `ResultsServlet` umzusetzen, die die Ergebnisseite entsprechend der eingangs dargestellten Skizze anzeigen kann. Die zu präsentierenden Daten sollen dem zentralen `Results`-Objekt entnommen werden.

Durch einen Klick auf den Knopf "Zurücksetzen" ist per HTTP-GET ein Formular an das `ResultsServlet` zu senden, welches nur ein verstecktes Feld mit dem Namen `reset` enthält. Der Inhalt dieses Feldes ist nicht relevant und kann daher willkürlich gewählt werden. Findet nun das Servlet in der HTTP-Anfrage einen Parameter vor, der ebendiesen Namen hat, dann sind die Inhalte des `Results`-Objekts zu löschen und alle im `ServletContext` gespeicherten Antworten auf Prüfungsfragen zu entfernen.

Prüfen Sie dabei, ob und wann eine Synchronisierung des `Results`-Objekts nötig ist.

**Teilaufgabe e**

Die durch das `ResultsServlet` generierte Ergebnisseite soll nun um einen "Download"-Knopf erweitert werden, über den der Benutzer in die Lage zu versetzen ist, die Inhalte der `Results`-Instanz als reine Textdatei herunterzuladen. Es soll ihm also eine textdateibasierte Auflistung zur Verfügung gestellt werden, über die erkennbar ist, wer auf die einzelnen Fragen mit "Ja" respektive mit "Nein" geantwortet hat.

Erstellen Sie eine eigene Servlet-Klasse mit dem Namen `DownloadServlet`, in der das Herunterladen der Ergebnisübersicht abgewickelt wird. Stellen Sie sicher, dass die Ergebnisseite schlussendlich genau zwei Formulare enthält: dasjenige zum Zurücksetzen der Resultate (siehe Aufgabenteil d) sowie das in diesem Aufgabenteil beschriebene.

**Teilaufgabe f**

Abschließend ist die Webanwendung um eine JSP-Seite zu erweitern, die dieselben Inhalte und Funktionen wie die Klasse `ExamServlet` bietet. Reichen Sie die JSP-Seite unter dem Namen `results.jsp` mit dem Rest Ihrer Lösung ein.

**Hinweise für ASB:**

- Bezogen auf Teilaufgabe b ist es ausreichend, wenn die von Ihnen eingereichte Lösung nur die unter Schritt 2) beschriebene Funktionalität enthält. Die Funktionalität von Schritt 1) wird seitens des ASB-Systems nicht verlangt.
- Alle Klassen sowie auch die Aufzählung müssen öffentlich sein und sich im Paket `da.tasks.servlet.exam` befinden.