Aufgabenstellung für das *erste* Übungsbeispiel der LVA "Objektorientiertes Programmieren"

Abgabeschluss für den Source Code: Sonntag 18.04.2021

WICHTIG:
Wird das Beispiel nicht fristgerecht abgegeben, haben Sie in Folge nicht mehr die Möglichkeit, die LVA im laufenden Semester erfolgreich abzuschließen!

Abgabegespräch:

Zwischen ***20.04.2021*** *und* ***22.04.2021*** (Jeweils zwischen 13:00 und 18:00 Uhr: Den Tag und die genaue Uhrzeit wählen Sie selbst in TUWEL.)

Allgemeines

Jeder Teilnehmer muss die Beispiele **eigenständig** ausarbeiten. Bei der Abgabe jedes Beispiels gibt es ein Abgabegespräch mit einem Tutor oder Assistenten. Diese Abgabegespräche bestehen aus den folgenden Punkten:

- Im Zuge der Abgaben müssen Sie eine Codeänderung vornehmen können.

  Achtung: Bei der Abgabe werden Autovervollständigung sowie automatische Fehlerkennzeichnung, Fehlerbehebung und Codevorschläge nicht verfügbar sein! Bereiten Sie sich entsprechend vor.

  Eine Anleitung, welche Eclipse Einstellungen dazu anzupassen sind, finden Sie in dem Dokument **Disabling the Code Assistance in Eclipse** in TUWEL.

- Beantworten von Fragen zu Ihrem Programm.
- Ihr Programm wird mit automatisierten Tests geprüft.
- Beantwortung **theoretischer Fragen** zu den jeweils in der Übung behandelten **Konzepten**.

Umsetzung und Abgabe

- Halten Sie sich exakt an die Beschreibungen der Klassen und deren Methoden.
- Geben Sie Ihrem Eclipse-Projekt einen eindeutigen Namen. Halten Sie sich dabei an folgende Benennungsvorschrift:

  Eclipse-Projekt: *matrNr_Beispiel1_Nachname*

  Beispiel: *00123456_Beispiel1_Mustermann*

  ACHTUNG: Wenn Sie sich nicht an die Benennungsvorschrift halten, kann Ihre Abgabe nicht gewertet werden!

- Name und Matrikelnummer muss zu Beginn jeder Klasse in einem Kommentarblock angeführt werden!
- Exportieren Sie das Projekt (inklusive aller für Eclipse notwendigen Dateien, z.B.: .classpath, .project) als ZIP-Datei. Eine Anleitung dazu finden Sie in TUWEL.

Kontakt:

Inhaltliche Fragen: Diskussionsforum in TUWEL

Organisatorische Fragen: oop@ict.tuwien.ac.at

Aufgabe: Reise Buchungssystem (RBS)

Ziel:

Erwerb konkreter Erfahrung und Kompetenz in der Umsetzung folgender Konzepte:

Vererbung, Casting, Polymorphismus, Overloading, Overriding, Exceptions, Enumerations.

Einleitung:

Für das Erreichen der Minimalanforderung im Praxis-Teil ist Voraussetzung, dass für alle Klassen die in den unten angeführten Methodenbeschreibungen definierten Anforderungen umgesetzt wurden. Um die zusätzlichen erreichbaren Bonuspunkte zu erhalten, muss die weiter unten angeführte Bonusaufgabe gelöst werden. Die Funktionalität Ihres Programms muss im eigenen Ermessen getestet werden.

Aufgabenbeschreibung

Mittels des Reise Buchungssystems (RBS) ist es möglich Reisen, im speziellen Flugreisen, zu buchen. Das RBS ermöglicht die Verwaltung von verfügbaren Flügen und Reisebüros (travel agencies). Zusätzlich können Buchungen (bookings) über das RBS durchgeführt werden. Jede Buchung von Flugreisen erfolgt dabei in einem Reisebüro. Es können dabei nur im RBS verwaltete Flüge gebucht werden. Eine Buchung wird dabei durch die Flugdaten, das Reisebüro und einen Buchungsstatus (booking status) beschrieben. Abhängig vom Buchungsstatus können Buchungen noch geändert werden oder nicht.

Die Minimalanforderung besteht darin das RBS entsprechend der Vorgaben zu implementieren.

Das RBS teilt sich in mehrere Java-Packages auf. Eine grobe Übersicht der im Gesamtsystem enthaltenen Packages finden Sie im Folgenden. Die detaillierte Beschreibung der Klassen und Interfaces entnehmen Sie der beiliegenden untenstehenden Methodenbeschreibungen. Implementieren Sie die untenstehenden Packages in der **Reihenfolge von unten nach oben.**

Packages

- **rbs:** In diesem Package werden die Klassen für das Buchungssystem, die Reisebüros und Buchungen implementiert.
- **rbs.flight:** Dieses Package bietet die Möglichkeit Flüge zu erstellen und Schnittstellen um auf diese zuzugreifen. Es gibt drei Arten von Flügen: *OneWayFlight*, *MultiStopFlight*, *RoundTripFlight*. Ein *MultiStopFlight* ist eine aus mehreren Flügen zusammengesetzte Flugreise deren Flüge so aufeinander abgestimmt sind, dass der Zielort (destination) des ersten Fluges der Ausgangspunkt (point of departure) des zweiten Fluges usw. sein muss. Ein *RoundTripFlight* besteht aus zwei Flügen bei denen der Ausgangspunkt des ersten Fluges mit dem Zielort des zweiten Fluges ident ist.
- **rbs.record:** Deklariert abstrakte Typen für Protokolleinträge.
- **ict.basics:** Dieses Package definiert ein Interface das für Klassen festlegt, dass Kopien Ihrer Instanzen erstellt werden können.

Beschreibungen:

Dieser Abschnitt enthält kurze Erläuterungen zu den einzelnen Klassen und Interfaces die in diesem Übungsbeispiel verwendet werden.

Die Klasse `BookingSystem` repräsentiert ein Buchungssystem, dass die Verwaltung der verfügbaren Flüge (Attribut: *flightPlan* vom Typ *List<IFlight>*), der Reisebüros (Attribut: *agencies* vom Typ *List<TravelAgency>*) und getätigten Flug-Buchungen (Attribut: *bookings* vom Typ *List<Booking>*) erlaubt.

Flüge werden im Buchungssystem mittels Klassen aus dem Package `rbs.flight` verwaltet. Das Buchungssystem unterstützt unterschiedliche Arten von Flügen, welche implementierungstechnisch in einer Hierarchie abgebildet sind. Jeder Flug wird durch eine Menge an Flugdaten beschrieben. Die Abfrage dieser Flugdaten ist über die Schnittstelle `IFlight` festgelegt. Um Gemeinsamkeiten von Flügen (wie etwa den Preis) nicht mehrfach zu beschreiben, wird eine abstrakte Klasse Flight eingesetzt. In dieser Klasse werden Basisfunktionalität und Attribute die alle Flüge gemeinsam haben verwaltet (z.B. Preis, Abflugs- und Zielort). Konkrete Ausprägungen von Flügen sind: einfache Flüge in der Klasse `OneWayFlight`, Hin- und Rückflüge in der Klasse `RoundTripFlight` und Flüge mit Zwischenstopps in der Klasse `MultiStopFlight`. Ein `OneWayFlight` besteht nur aus einem Flug der von einem Abflug- zu einem Zielort geht. Ein Hin- und Rückflug besteht aus zwei Flügen bei denen der Zielort des Rückflugs ident mit dem Abflugort des Hinflugs ist. Zusätzlich ist der Zielort des Hinflugs ident mit dem Abflugort des Rückflugs. Ein Flug mit Zwischenstopps kann beliebig viele Flüge beinhalten wobei der Abflugort des nächsten Fluges immer gleich dem Zielort des vorherigen Fluges sein muss. Wird ein neuer Flug hinzugefügt, muss sichergestellt werden, dass diese Eigenschaft nicht verletzt wird. Ist dies Eigenschaft nicht erfüllt, wird eine Fehlermeldung über die Exception `FlightsNotConnectedException` ausgelöst. Diese Fehlermeldung beinhaltet alle Flüge die zu dieser Verletzung geführt haben.

Buchungen werden durch die Klasse `Booking` repräsentiert. Eine Buchung besitzt eine eindeutige Buchungsnummer (Attribut: *id* vom Typ *long*), ist einem Reisebüro (Attribut: *agency* vom Typ *TravelAgency*) zugeordnet und enthält eine Liste an Flügen (Attribut *flights* vom Type *List<IFlight>*), die gebucht wurden. Zusätzlich besitzt eine Buchung einen Status (Attribut *state* vom Type *BookingState*) der Aufschluss darüber gibt wie weit die Buchung fortgeschritten ist. Der Buchungsstatus wird durch die Enumeration `BookingState` repräsentiert. Eine Buchung kann dabei die folgenden Zustände besitzen: `OPEN` (Buchung kann noch verändert werden), `CLOSED` (alle Buchungsdaten sind eingegeben, keine Änderungen mehr möglich aber noch nicht bezahlt), `PAID` (Buchung wurde bezahlt) und `CANCELLED` (Buchung wurde storniert).Reisebüros werden durch die Klasse `TravelAgency` repräsentiert. Jedes Reisebüro hat einen Namen (Attribut: *name* vom Typ *String*) und einen Rabatt (Attribut: *discount* vom Typ *float*) in Prozent der bei Buchungen über das jeweilige Reisebüro gewährt wird. Die vorhandenen und buchbaren Flüge sind als Attribut im Buchungssystem hinterlegt. Kann eine Buchung eines Fluges nicht durchgeführt werden, tritt eine Fehlermeldung repräsentiert durch die Exception `NotBookableException` auf.

Um sicherzustellen, dass gebuchte Flüge nicht nachträglich geändert werden können, beispielsweise durch eine Anpassung des Preises, werden nur Kopien der Flüge in den Buchungen eingetragen. RBS bietet einen Kopiermechanismus für alle Klassen in `rbs.flight` über die Methode deepCopy an. Jede Klasse, in RBS jeder Flug, der kopiert werden kann, ist durch ein Interface `IDeepCopy` gekennzeichnet.

Implementierungsfahrplan (Hinweis/Vorschlag):

In diesem Abschnitt möchten wir Ihnen einen Hinweis bezüglich der Reihenfolge in der Sie die Klassen und Interfaces implementieren sollten geben. Ziel ist es mit jenen Teilen anzufangen, welche die geringsten Abhängigkeiten von anderen aufweisen.

Vorgeschlagene Reihenfolge:

1. IDeepCopy
2. IFlight
3. FlightsNotConnectedException
4. Flight, OneWayFlight, RoundTripFlight, MultiStopFlight
5. IRecord
6. Record
7. BookingState
8. Booking
9. TravelAgency
10. NotBookableException
11. BookingSystem

Interface-, Klassen- und Methodenbeschreibungen:

In diesem Abschnitt finden Sie detaillierte Beschreibungen zu einzelnen Interfaces, Klassen und deren Methoden.

Notationen:

- Package: Das jeweilige Package steht vor dem Klassen- bzw. Interface-Namen. Zum Beispiel besagt **rbs.record**::IRecord, das sich das Interface `IRecord` im Package `rbs.record` befindet.
- Attribute: Die Notation bezüglich Attribute folgt jener in Klassendiagrammen (Diagramme die wir später kennen lernen werden) und geben Aufschluss über **Sichtbarkeit, Namen und Typ** des Attributs.
- Methoden: Die Notation bezüglich Methoden folgt ebenfalls jener in Klassendiagrammen und gibt Aufschluss über **Sichtbarkeit, Modifier, Name, Parameterliste, Rückgabetyp und (checked) Exceptions** die von der Methode geworfen werden können.
- Sichtbarkeiten sind bei Attributen und Methoden dabei folgendermaßen codiert:

  `+ … public` `# … protected` `- … private`

- Klassenvariablen und statische Methoden sind unterstrichen dargestellt.
- Abstrakte Methoden sind kursiv dargestellt.
- Die Exception die von einer Methode geworfen werden kann, wird nach dem Keyword **throws** angeführt.

Beispiele zur Notation:

`- agencies:List<TravelAgency>`

Notation für eine private Klassenvariable (static) mit dem Namen *agencies* vom Typ *List<TravelAgency>* (Liste von Objekten vom Typ *TravelAgency*)

`+ bookFlight(flight:IFlight):long throws MyException`

Notation für die öffentliche (public) Methode bookFlight mit dem Übergabeparametertyp *IFlight* und dem Rückgabetyp *long*. Zusätzlich kann die Methode eine (checked) Exception vom Typ MyException werfen.

rbs::BookingSystem

Bei BookingSystem handelt es sich um eine **Klasse**.

Attribute:

- `agencies:List<TravelAgency>`
- `flightPlan:List<IFlight>`
- `bookings:List<Booking>`

Methoden:

`+ createAgency(name:String):void`

Legt eine neue Agency im Buchungssystem an und fügt sie der List *agencies* hinzu.

`+ getAgencyNames(): List<String>`

Gibt alle Namen der Agencies in der List *agencies* in einer List zurück.

`+ findAgency(name:String):TravelAgency`

Sucht in der List *agencies* die Agency mit dem entsprechenden Namen und gibt diese zurück. Wird keine Agency mit diesem Namen gefunden soll `null` zurückgegeben werden.

`+ findAgency(agency:TravelAgency):TravelAgency`

Sucht in der List *agencies* diejenige Agency die inhaltlich gleich ist. Zwei Travelagencies werden dann als inhaltlich gleich angesehen, wenn `a1.equals(a2) == true`. Wird keine solche Agency in der Liste gefunden soll `null` zurückgegeben werden.

`+ addFlight(flight:IFlight):boolean`

Fügt einen Flug in die List *flightPlan* ein. Ein Flug darf nur hinzugefügt werden, wenn sich noch kein Flug mit derselben Id in der Liste befindet. Falls das Einfügen erfolgreich war, soll `true` zurückgegeben werden, ansonsten `false`.

`+ addFlight(flights:List<IFlight>): boolean`

Fügt alle in *flights* enthaltenen Flüge in die List *flightPlan* ein. Kann ein Flug nicht hinzugefügt werden, so wird er übersprungen und mit dem nächsten Eintrag in der Liste fortgesetzt. Wenn alle Flüge erfolgreich hinzugefügt werden konnten, wird `true` zurückgegeben, ansonsten `false`.

`+ containsFlight(flight:IFlight):boolean`

Gibt `true` zurück wenn sich der Flug in der Liste *flightPlan* befindet. Die Überprüfung auf Gleichheit erfolgt mittels equals.

```
+ bookFlight(flight:IFlight, agency:TravelAgency):long
                                        throws NotBookableException
```

Bucht den Flug für die Agency und legt eine neue Buchung (booking) an. Der Flug wird nur dann gebucht, wenn er sich auch in der Liste *flightPlan* befindet. Falls der Flug gebucht werden kann wird die Id der Buchung zurückgegeben, ansonsten wird eine `NotBookableException` geworfen.

```
+ bookFlight(flights:List<IFlight>, agency:TravelAgency):long
                                        throws NotBookableException
```

Bucht alle Flüge in der List *flights* für die Agency. Die Buchung wird nur angelegt, wenn **alle** Flüge gebucht werden können. Ein Flug kann nur dann gebucht werden, wenn er sich auch in der Liste *flightPlan* befindet. Die Einträge in die Liste *flights* der Buchung (Booking) werden dabei als Kopien gespeichert. Konnte die Buchung erfolgreich durchgeführt werden so wird die Id der Buchung zurückgegeben, ansonsten wird eine NotBookableException geworfen.

`+ addFlightToBooking(flight:IFlight, bookingId:long):boolean`

Fügt einem Booking mit der Id den übergebenen Flug hinzu. Falls keine Buchung mit der Id existiert oder diese sich nicht im Status OPEN befindet wird abgebrochen und `false` zurückgegeben. Der Flug kann nur dann gebucht werden, wenn er sich auch in der Liste *flightPlan* befindet. Falls der Flug gebucht werden kann wird `true` zurückgegeben, ansonsten `false`.

`+ getFlights():List<IFlight>`

Gibt die Liste *flightPlan* zurück.

`+ getBookingById(id:long):Booking`

Gibt das Booking mit der entsprechenden Id zurück.

`# generateReturnFlight(flight:OneWayFlight):IFlight`

Generiert aus einem OneWayFlight einen neuen OneWayFlight, mit getauschtem Ausgangspunkt und Zielort. Der Preis wird übernommen. Die *flightId des* Rückfluges wird durch anhängen von *R* an die *flightId* des Hinfluges erzeugt. Zusätzlich wird der Rückflug in die Liste *flightPlan* gespeichert, falls noch kein Flug mit dieser ID darin enthalten ist. Ist dies erfolgreich, wird der Retourflug zurückgegeben, ansonsten `null`.

`- generateOneWayFlights():List<IFlight>`

Gibt eine Liste mit mindestens 5 `OneWayFlights` zurück.

`- generateMultiStopFlights():List<IFlight>`

Gibt eine Liste von mindestens 3 MutliStopFlights zurück. Wobei mindestens ein `MultiStopFlight` einen anderen `MultiStopFlight` beinhaltet.

`- generateRoundTripFlights():List<IFlight>`

Gibt eine Liste von mindestens 3 `RoundTripFlights` zurück.

`+ main(args:String[]):void`

Die Main-Methode ist der Einstiegspunkt in das Programm und sollte folgende Anforderungen erfüllen:

- Erzeugen eines Flugplans bestehend aus
  - Mindestens 5 `OneWayFlights`
  - Mindestens 3 `MultiStopFlights`.
  - Mindestens 1 `MultiStopFlights` die mindestens einen anderen `MultiStopFlight` beinhalten.
  - Mindestens 3 `RoundTripFlights`
- Versuch einen bereits im Flugplan vorhandenen Flug ein zweites Mal hinzuzufügen
- Anlegen von mindestens 2 `TravelAgencies`.
- Buchungen durchführen
  - Jede Methode mit der eine Buchung erstellt werden kann muss mindestens einmal aufgerufen werden.
  - Einer bestehenden Buchung muss ein neuer Flug hinzugefügt werden.
  - Zu jeder `TravelAgency` muss es mindestens 2 Buchungen geben.
- Umsetzung eines Consolen-basierten Menüs:
  - Suchen eines Flugs anhand der Flugnummer (flightId) und anschließende Ausgabe der Flugdaten
  - Hinzufügen eines neuen Fluges (`OneWayFlight`)

rbs.record::IRecord

Bei IRecord handelt es sich um ein **Interface**.

Methoden:

`+ getId():long`

Gibt die Id eines Records zurück.

rbs.record::Record

Die **abstrakte Klasse** implementiert das Interface IRecord.

Attribute:

`- id:long`

Konstruktor:

`+ Record(id:long)`

Der Konstruktor setzt das Attribut id.

Methoden:

`+ getId():long`

Implementiert die Methode getId() aus IRecord.

rbs::Booking

Die **Klasse** Booking leitet von der Klasse Record ab.

Attribute:

- `flights:List<IFlight>`
- `agency:TravelAgency`
- `state:BookingState`
- `uniqueBookingId:long`

Konstruktor:

`+ Booking(id:long, agency:TravelAgency, flights:List<IFlight>)`

Der Konstruktor setzt die Attribute. Wobei die Flüge in *flights* als Kopie in die Buchungsliste eingetragen werden und der Status auf OPEN gesetzt wird.

Methoden:

`+ getFlights():List<IFlight>`

Gibt eine Liste der gebuchten Flüge zurück.

`+ addFlight(flight:IFlight):void`

Fügt den übergebenen Flug *flight* der Liste *fights* als Copy hinzu.

`# setState(state:BookingState):boolean`

Setzt den State des Bookings. Ein Übergang von OPEN zu PAID ist nicht erlaubt. Gibt `true` zurück, wenn der State gesetzt werden kann, ansonsten `false`.

`+ getState():BookingState`

Gibt den State der Buchung zurück.

`+ isPaid():boolean`

`+ getAgency():TravelAgency`

`+ generateBookingId():long`

Gibt den Wert von *uniqueBookingId* zurück und erhöht ihn um 1.

---

rbs::BookingState

Bei BookingState handelt es sich um ein **Enum** das folgende Konstanten abbildet.

`OPEN`

`CLOSED`

`PAID`

`CANCELLED`

---

rbs::TravelAgency

Bei TravelAgency handelt es sich um eine **Klasse** mit folgenden Attributen und Methoden.

Attribute:

`- name:String`

`- discount:float`

Konstruktoren:

`+ TravelAgency(name:String, discount:float)`

Der Konstruktor erstellt eine neue Instanz und setzt die Attribute *name* und *discount*.

`+ TravelAgency(name:String)`

Der Konstruktor erstellt eine neue Instanz und setzt das Attribut *name.* Als Standard-Wert für das Attribut *discount* wird 0 verwendet.

Methoden:

`+ getDiscount():float`

`+ setDiscount(discount:float):void`

`+ getName():String`

`+ toSting():String`

`+ equals(obj:Object):boolean`

Eine TravelAgency ist gleich zu einem Objekt *obj* wenn sie vom gleichen Typ sind und die Namen gleich sind.

ict.basics::IDeepCopy

Das **Interface** IDeepCopy definiert die folgende Methode.

Methoden:

`+ deepCopy():IDeepCopy`

Erlaubt die Erzeugung einer Kopie eines Objektes. Dabei sollen von allen referenzierten Objekten von Klassen, die das IDeepCopy Interface implementieren, ebenfalls Kopien angelegt und diese Kopien referenziert werden.

Nach Ausführung dieser Methode gilt für ein Objekt x - `x.deepCopy() != x`.

rbs.flight::IFlight

Das Interface IFlight erweitert das Interface IDeepCopy und definiert darüber hinaus folgende Methoden.

Methoden:

```
+ getFlightId():String
+ getPrice():float
+ getDestination(): String
+ getDeparture():String
+ deepCopy():IFlight
```

rbs.flight::Flight

Abstrakte Klasse, die das Interface IFlight implementiert.

Attribute:

```
- flightId:String
# departure:String
# destination:String
- price:float
```

Konstruktoren:

`+ Flight(id:String, departure:String, destination:String,price:float)`

Der Konstruktor setzt die Attribute.

`+ Flight(id:String, departure:String, destination:String)`

Der Konstruktor setzt die Attribute. Der Standardwert für das Attribut *price* ist mit 100 festgelegt.

`# Flight(id:String)`

Setzt das Attribut *id* sowie destination = "" und departure = "".

Methoden:

```
+ getFlightId():String
+ getPrice():float
+ getDestination(): String
+ getDeparture():String
+ toString():String
```

Liefert eine String-Repräsentation der Fluges in der Form *Flight[attribute1=Attribute1Value,attribute2=Attribute2Value...]*

`+ setPrice(price:float):void`

rbs.flight::OneWayFlight

Die **Klasse** OneWayFlight leitet von der abstrakten Klasse Flight ab.

Konstruktoren:

```
+ OneWayFlight(id:String, departure:String, destination:String)
+ OneWayFlight(id:String, departure:String, destination:String, price:float)
```

Methoden:

```
+ equals(obj:Object):boolean
```

Ein Flight ist gleich zu einem Objekt *obj* wenn sie vom gleichen Typ sind, die flightId gleich sind, sowie der Ausgangspunkt und der Zielort gleich sind.

```
+ deepCopy():OneWayFlight
```

rbs.flight::MultiStopFlight

Die **Klasse** MultiStopFlight leitet von der abstrakten Klasse Flight ab.

Attribut:

```
- flights:List<IFight>
```

Konstruktoren:

```
+ MultiStopFlight(id:String, flight:IFlight)
```

Der Konstruktor legt einen neuen MultiStopFlight an, setzt die Id und trägt den Flug als Kopie in das Attribut flights ein.

```
+ MultiStopFlight(id:String, flights:List<IFlight>)
                                    throws FlightsNotConnectedException
```

Der Konstruktor setzt die Id und trägt die übergebenen Flüge als Kopie in das Attribut *flights* ein. Dabei erfolgt die Eintragung nur wenn die Flüge nahtlos ineinander übergehen, also wenn der Zielort des vorherigen Fluges gleich dem Startpunkt des nächsten Fluges ist. Ist dies nicht der Fall wird eine `FlightsNotConnectedException` geworfen. Das Attribut destination wird mit dem neuen Zielort aktualisiert.

Methoden:

```
+ getPrice():float
```

Berechnet dynamisch den Preis des gesamten MultiStopFlights. Der Preis ergibt sich dabei aus der Summe der Preise der einzelnen Flüge.

```
+ getFlights():List<IFlight>
+ addFlight(flight:IFlight):void throws FlightsNotConnectedException
```

Fügt den Flug *flight* als Kopie in die Liste *flights* ein. Der Flug wird nur hinzugefügt, wenn er sich nahtlos in die Flugreise eingliedert. Dazu wird überprüft ob der letzte Zielort des letzten Fluges in der Liste mit dem Ausgangspunkt des hinzuzufügenden Fluges übereinstimmt. Wenn der Flug nicht hinzugefügt werden kann, wird eine `FlightsNotConnectedException` geworfen. Das Attribut destination wird mit dem neuen Zielort aktualisiert.

```
+ equals(obj:Object):Boolean
```

Eine MultiStopFlight ist gleich zu einem Objekt *obj* wenn sie vom gleichen Typ sind, die FlightId gleich ist und die Fluglisten übereinstimmen.

```
+ toString():String
```

Liefert eine String-Repräsentation des MutiStopFlights und dessen Teilflügen in geeigneter Form wieder.

`+ deepCopy():MultiStopFlight`

Sollte das Kopieren bzw. neu Anlegen eines MultiStopFlights fehlschlagen (`FlightsNotConnectedException`), wird `null` zurückgegeben.

rbs.flight::RoundTripFlight

Die **Klasse** RoundTripFlight leitet von der abstrakten Klasse Flight ab.

Attribut:

`- flights:List<IFights>`

Konstruktor:

`+ RoundTripFlight(id:String, flights:List<IFlight>)`

Der Konstruktor setzt die Id und trägt die Flüge als Kopien in das Attribut *flights* ein. Die Flüge werden dabei nur übernommen, wenn sie die Bedingungen an einen Roundtrip also Ausgangsort des ersten Fluges ist die Destination des zweiten Fluges, erfüllen. Als Ausgangsort und Destination des RoundTripFlights werden jeweils Ausgangsort und Destination des Hinfluges gesetzt.

Methoden:

`+ getPrice():float`

Berechnet dynamisch den Preis des gesamten RoundTripFlights, abzüglich eines fixen Roundtrip-Discounts von 10%.

`+ getFlights():List<IFlight>`

`+ equals(obj:Object):Boolean`

Eine RoundTripFlight ist gleich zu einem Objekt *obj* wenn sie vom gleichen Typ sind, die FlightIds gleich sind und die Fluglisten übereinstimmen.

`+ toString():String`

Liefert eine String-Repräsentation des RoundTripFlight und dessen Teilflügen in geeigneter Form wieder.

`+ deepCopy():RoundTripFlight`

rbs.flight::FlightsNotConnectedException

Die **Klasse** FlightsNotConnectedException leitet von der Klasse Exception der Java-API ab.

Attribute:

`- flights:List<IFlight>`

`+ final serialVersionUID:long = 0l;`

Konstruktor:

`+ FlightsNotConnectedException(flights:List<IFlight>)`

Der Konstruktor speichert die übergebene Liste in das Attribut *flights*.

Methoden:

`+ getMessage():String`

Gibt eine Repräsentation der Fehlermeldung zurück, die die Flights in geeigneter Form enthält.

rbs::NotBookableException

Die **Klasse** NotBookableException leitet von der Klasse Exception der Java-API ab.

Attribut:

```
+final serialVersionUID:long = 2l;
```

BONUSAUFGABE

In Ihrer implementierten Version des RBS gibt es aktuell keine Möglichkeit Daten zu persistieren. Das bedeutet, dass alle Daten, wie etwa Flüge, bei jedem Start von RBS programmatisch oder per Konsole neu eingetragen werden müssen. Eine Speicherung von Flügen sowie ein automatisches Laden bestehender Flüge aus einer Datenquelle wäre wünschenswert. Zu diesem Zweck müssen die Daten (Objekte) serialisiert werden. Unter serialisieren versteht man die Abbildung von strukturierten (Klassen/Objekte) Daten auf eine sequentielle Darstellungsform.

Aufgabenstellung

Schaffen Sie eine Möglichkeit Instanzen von `OneWayFlights` lokal, in einer Datei auf dem Dateisystem, zu persistieren und auch wieder von dort zu laden. Verwenden Sie für die Serialisierung das JSON-Datenformat[1]. JSON Daten werden durch key-value Paare beschrieben:

```
{
      "flightId":      "Wien-Berlin"
      "price":          235.0

      …
}
```

Erstellen Sie eine neue Klasse `JSONSerializer` mit einer `main`-Methode. Implementieren Sie zwei Methoden `serializeJSON` und `deserializeJSON` die es ermöglichen, Objekte des Typs OneWayFlight in JSON zu konvertieren bzw. JSON in ein Objekt des Typs OneWayFlight zu konvertieren. Da die Serialisierung von polymorphen Klassen nicht trivial ist, ist dieses Beispiel auf die Klasse OneWayFlight beschränkt.

Verwenden Sie für Ihre Implementierung die Library GSON[2]. GSON erlaubt es Datentypen relativ einfach nach JSON zu konvertieren. UM GSON verwenden zu können, müssen Sie lokal eine Bibliothek einbinden, d.h. zum Buildpath/Classpath hinzufügen. Das passende Jar-File finden Sie im Abgabe Zip-File im Ordner Bonus. Ein einfaches Beispiel für die Programmierung mit GSON finden Sie unter folgendem Link: https://www.tutorialspoint.com/gson/gson_first_application.htm

Sie können auch Listen von Objekten mit GSON serialisieren. Wollen Sie diese Daten allerdings wieder laden, müssen Sie den entsprechenden Typ der Liste mit angeben:

```
Type tL = new TypeToken<ArrayList<OneWayFlight>>() {}.getType();
```

Legen Sie eine Liste mit mehreren OneWayFlights an und speichern Sie diese im JSON Format in einer Textdatei. Anschließend laden Sie den Inhalt dieser Datei und erzeugen die Liste mit den OneWayFlights erneut. Geben Sie abschließend das Ergebnis aus.

[1] https://de.wikipedia.org/wiki/JavaScript_Object_Notation
[2] https://github.com/google/gson