Programming 2

Christian Grévisse (christian.grevisse@uni.lu)

Lab 9 – OOP Basics & Closures

2018–04–26

Exercise 30 – Airport

Similar to exercise 26, write a Swift console application that models a basic airport management system.

1° Create the classes `Airport` and `Airline`. Both[1] contain *stored properties* for their name and IATA code (3 letters[2] for airports, 2 letters for airlines).

2° Create a class `Passenger` which holds the name and email address of a passenger.

3° Create a class `Flight` that holds properties for the airline operating the flight, the flight number, the origin and destination airports, the time (arrival or departure) and an *optional* gate, initially set to `nil`. You may model the time either as a tuple of two integers, or define a separate class (for the nerds: have a look at the `Date` and `DateComponent` types from the `Foundation` library).

- Add a *computed property* `flightDesignator` which comprises the 2-letter IATA code of the airline and the flight number. For instance, a Luxair (IATA: `LG`) flight with flight *number* (stored property) 601 has the flight *designator* (computed property): `LG601`.
- Add a seat map representing the allocation of seats, i.e. a seat can either be assigned to a `Passenger` or left empty (`nil`). How to use multidimensional arrays in Swift is explained in the Language Reference[3].
- Create both a *designated* and a *convenience* initializer.
  - The designated initializer has parameters for all the properties. The seat map is initialized using two parameters for the number of rows and seats per row. These parameters are given with a default value. These default values are stored as constant *type properties*.
  - The convenience initializer only receives parameters for airline, flight number, origin and destination airports, and the time.
- Add property observers for both the time and gate properties. Both observers shall call, for all passengers in the seat map, the method `notifyFlightUpdate(flight:Flight, message:String)` of the `Passenger` class. This method simulates an email sent to a passenger with a message according to the flight change. When the time is changed, the old and new value are given in the message. When the gate is set for the first time (from `nil` to a non-`nil` value), the message contains the gate announcement. If the gate later on changes, the message shows the old and new gate.
- Add a method `checkin(passenger p:Passenger) -> (Int, Int)?` which performs the check-in for a passenger on the flight. The method returns the "coordinates" (row, seat) of the first available seat, if any.
- Add methods to print the seat map, the passenger list and the relevant flight information.

4° Create a class `AirportManager`, which contains properties for the administered airport and the schedule, which is an array of flights.

- Add a method to add flights to the schedule.
- Add a method `printSchedule()` that prints the schedule. First, print the departing flights, then the arriving ones. You may rely on the `filter` method defined for array types. This method expects a *(trailing) closure* that indicates the condition for an element to be included in the resulting array. Extract the printing logic in a separate method `printFilteredFlights(filter: (Flight) -> Bool)` which receives a filter closure as a parameter, applies it on

[1] Obviously, this repetition can be avoided once we will have seen inheritance in Swift.

[2] You might enjoy the `prefix` method that can be applied on `String` instances (requires `import Foundation` on Linux)

[3] Have a look at the `Array(repeating:count:)` initializer, too.

the schedule and prints the information of the resulting array of flights. This way, the `printSchedule()` method only needs to call former method with an appropriate filter condition.

- Add a property observer to the schedule property, such that the schedule is printed each time the schedule changes (addition of a flight).

5° Test your classes by creating an airport manager, adding a couple of flights, checking a passenger in, printing the seat map and passenger list, changing the gate and flight time, etc. A menu as in exercise 26 is not necessary. However, the `readLine(strippingNewline: Bool = default) -> String?` function could be useful, if you want to write one. Some boilerplate code as well as an example output are provided on Moodle.

General advice: Use `let` and access control wisely!