CMPT 270 – 2016-2017 Term 01

# Assignment 2:

Getting Started with Java

**Submission Deadline:**
**Part A: 28 September 2016 at 11:55pm – Submit via Moodle!**
**Part B: 5 October 2016 at 11:55pm – Submit via Moodle!**

## Description

The objective of this assignment is to get familiar with the basics of the Java language, and the Java system. To do this, you will write the classes to model three entities of part of a flight reservation application, and test them.

In particular, you must create the three classes: a `Person` class, a `Booking` class, and a `BasicFlight` class. Each class should have a constructor, and the fields and methods specified below. For each of the classes, all the fields should be declared as private. Be sure to follow the specification given below, since the classes will be used in subsequent assignments to build a simple flight reservation system. In addition, each class is to have a static main method that should be used to test the class following the principles of regression testing as discussed in class.

## Deliverables

***** Upload .java files, .class files and .txt files for documentation (no Word .docs) *****

### Part A:

The `Person` class is the simplest class, and it does not depend upon the others. Therefore, it will be done first for Part A of the assignment. We will get it marked as quickly as possible, so that you can correct any faults in it and eliminate any similar faults in the other classes before handling them all in Part B.

For Part A, upload a .java file of your `Person` class and the output obtained from the execution of your main method. If your output is a blank page, then be sure to clearly state that the output is a blank page. External documentation is not required for Part A, but proper internal documentation is required (including proper javadoc comment).

**Part B:**

For Part B, upload .java and .class files for all three classes, with the `Person` class revised in any ways that seem appropriate after its initial version has been marked, and it has been integrated with the other classes. Each class should be well written as described above. Also, you are expected to do a good job of testing all routines, including any additional tests for the `Person` class that you think should be added.

We may run your classes to verify that they run as you state. You should also include .txt files for:

(i) the external documentation for the assignment

(ii) a listing of all the classes,

(iii) a listing of the output from running your classes

The markers will provide written feedback on your work.

## Flight Reservation Application

Each of the classes are now discussed in more detail:

### `Person` class:

As would be expected, this class models a person. For our purposes, a person will have a name and a telephone number, where the telephone number might change but the name is not allowed to change. Thus the following features are required:

- name, a String field to store the person's name
- telephoneNumber, a String field to store the person's telephone number
- Person, a constructor with parameters for the person's name and telephone number
- getName, an accessor to return the person's name
- getTelephoneNumber, an accessor to return the person's telephone number
- setTelephoneNumber, a mutator with a string parameter to change the person's telephone number to the value of the string
- toString, the method to return a string representation of all the information about the person in a form suitable for printing; if printed, it should all print on the current line
- main, the method to test the above features

### `Booking` class:

This class is the representation for a booking of a person on a flight. The booking will also record the seat assigned to the person on the flight, if one has been assigned. The features for a booking are

- person, a Person field to store the person with the booking
- flight, the flight for the booking
- seat, a String field to store the seat; the seat is to have the format usually used by a flight, e.g., 13A
- Booking, a constructor with the Person and the BasicFlight for the booking as parameters; it assumes that the Person has not yet been assigned a seat
- three accessor methods with one accessor to access each of the fields

- setSeat, a mutator method to set the seat of the booking; the one parameter is a string storing the seat
- toString, the method to return a string representation of the information related to the booking, including the name of the person, the number of the flight, and the seat (only if a seat has been assigned); if printed, it should all print on the current line.
- main, the method to test the above features

`BasicFlight` **class:**

This class represents a very basic flight. The flight class should store the bookings for the people who have a spot on the flight. Obviously, a container is needed to store the bookings. Use an array for this task. The number of seats in a row is also stored (assuming that all rows have the same number of seats). The features are:

- number, a field to store the number of the flight; assume that it is an int value
- width, the number of seats per row
- bookings, the array to store the bookings for this flight, where each location of the array corresponds to a seat
- BasicFlight, a constructor with parameters for the number, width and total capacity of the flight
- accessor methods to obtain the number, width and capacity of the flight
- setSeat, a mutator method to store a booking in a seat of the flight, where the parameter is a booking that stores the person, this flight and the seat for the flight
- getPerson, an accessor method with a string parameter to obtain the person who booked the seat as specified by the string parameter, assuming that the seat has been booked
- isSeatAvailable, an accessor method with a string parameter to return a Boolean value that indicates whether the seat specified by the string is booked or not
- indexFromSeat and SeatFromIndex, see below
- toString, the method to return a string representation of the flight; in addition to the flight number, capacity and width (on one line), for each seat there should be a line with the seat name, and either nothing if the seat is not booked, or the name of the person who has a booking for the seat

- main, the method to test the above features

## Additional Guidelines

### Seats

Seats are specified as a String in the format of an integer followed by a capital letter. The capital letter must be one of the first w capital letters, where w is the width of the flight. Thus, the seats in row 3 for a flight with width 4 are 3A, 3B, 3C, and 3D. The first row is row 1, not 0. The bookings are to be stored in the array in the order first row, second row, third row, ..., and last row. Each row is stored in the order A, B, C, etc. Thus, it is necessary to be able to determine the index that corresponds to a seat string, and to determine the seat string that corresponds to an index. Include in the BasicFlight class methods to do these two conversions: indexFromSeat and SeatFromIndex. To assist you in doing these conversions, note that page 86 in Chapter 4 of Tremblay and Cheston has a list of methods for Strings (that might be useful). Also, the function Integer.parseInt with a String parameter parses a string to return the int stored in the string. However, be aware that the int must be the only thing in the whole string being parsed, including no leading or trailing blanks. In the handling of the char type, note that a char is actually a special notation for an integer value. For example, the char 'A' represents the value 65. Rather than trying to remember the number of each letter, you can use the fact that the capital letters have consecutive numbers, integer arithmetic can be done on char values, and an integer can be cast to a char value. Thus, (char)('A'+3) yields the char 'D'. Finally, note that a char value can be concatenated onto a String by the + operator. To simplify things, you can assume that every string specified for a seat on a flight correctly specifies a seat on the flight.

### Narrow scope of this assignment

The above classes do not do much – they are simply entity classes for use in a larger system. A flight reservation application would have a lot more functionality, such as:

- to obtain information and requests from users,
- to determine what operations are to be done,
- to carry out those operations by invoking methods of the entity classes,
- to return appropriate information back to the user.

Later assignments will fill in these parts. You do not need them for this assignment.

**Internal documentation**

Proper internal documentation includes:

- A comment just before the class header that gives an overview of the class. For example, if the class models an entity, the comment might state what entity the class models and specify the key features. Alternatively, if the class serves as a container, state what type of items the container stores, and how these items are accessed. Inside the body of the class, a comment might state what type of container it is (stack, queue, list, keyed or non-keyed dictionary, etc), and what representation is used to store the items. If the class is an interface, state for what it provides an interface and is there anything special about the interface. Finally, if it has a control function, what is it doing and controlling? Recall that comments for a class appear before the class and begin with /**
- A comment just before each field stating what is stored in the field, again beginning with /**
- A comment, beginning with /**, just before each constructor and each method stating what it does. Note that the comment only gives what the routine does, not how it does it. If it isn't obvious from the code how it accomplishes its goal, comments on how it is done belong in the body of the routine.
- Each class should be easily read. This includes good use of white space, especially reasonable indentation. The code should be easy to read in printed form as well as on a screen.
- Local variables that are simply used as temporary variables, and parameters that are just assigned to fields can be short, but other variables should have good descriptive names. The name should take into account of its location. For example, in the class Person it does not make sense to call a field personsName, since the field is in the Person class so the name would obvious be the person's name. Hence, simply call the field name. Be sure that if a variable is used as a local variable, it is declared as a local variable, not as a field. The only data stored in fields should be data that is intrinsically part of the object that needs to be stored from one use of the object (method invocation) until the next use.
- Although it isn't relevant for this assignment, no routine should be very long. An exception is a test routine. Test routines are often simply a long sequence of simple tests, so they might be longer.
- Every class should have a main routine to test the class. There should not be input statements. Instead, all data should be hard coded, so that you know

and can check the results. Be sure to invoke every method to ensure that it is behaving correctly. When testing one class, assume that other classes that it uses are correct. Therefore, you can concentrate on testing one class at a time. The testing should have the properties of regression testing as discussed in class. In particular, the tests for a class should be in its main routine, and the output should mostly only report errors. If there is any non-error output, the output should be checkable by reading the output without referring to the test routine. This is best done by having the output should state what is being output. For example,

The printout for Pete with number 249-5418 is

```
Name:  Pete Telephone number:  249-5418
```

**External documentation**

For external documentation, include the following:

- A description of how to execute your tests of the classes. What is necessary to compile your classes? What program or programs need to be run to show the execution of your program? For example, it might be necessary to compile and execute all three classes individually. On the other hand, you might have written a driver program (another class with a static main method that invokes the main methods of the three required classes; the static main method of class A can be invoked by A.main(null);). In the latter situation, we need to know the name of the driver class. This description should be very short.

- The status of your assignment. What is working and what is not working? What is tested and what is not tested? If it is only partially working, the previous point should have described how to run that part or parts that work. For the part or parts not working, describe how close they are to be working. For example, some of the alternatives for how close to working are (i) nothing done; (ii) designed but no code; (iii) designed and part of the code; (iv) designed and all the code but anticipate many faults; or (v) designed and all the code but with a few faults; (vi) working perfectly and thoroughly tested.

**Marking (total 40)**

Part A.1 (3) `Person` class

Part B.2 (4) `Booking` class

Part B.3 (10) `BasicFlight` class

Part B.4 (8) Testing code, proper regression testing

Part B.4 (3) Correctly functioning of all classes

Part B.5 (7) Clarity and quality of code style, internal documentation, submission organization.

Part B.6 (5) Clarity and quality of external documentation.

This is an individual assignment. You are encouraged to discuss the general concepts of classes, types, containers, etc. with you classmates, but the specific details of the Flight Reservation System in this assignment should be done completely individually. Students that copy / share work will be penalized.