Version Number:
1.01 (March 19, 2020)

Assignment 3

Room Booking Scheduler, Part 3

To be submitted online not later than Thursday, March 26rd, 2020, 11:59 p.m.

Description:

In this lab you'll continue to add new features to the RoomBooking Scheduler you started in Assignment 1. Along the way, you'll demonstrate your understanding of the following course learning requirements (CLRs), as stated in the *CST8284—Object Oriented Programming (Java)* course outline:

1. Use simple existing generic classes to manage objects, including searching and sorting using comparators (CLR V)
2. Write java code to implement exception handling (CLR VI)
3. Produce code that has been tested and executes reliably (CLR VIII)
4. Prepare program documentation using prescribed program specifiers, including JavaDoc (CLR IX)
5. Debug program problems using manual methods and computerized tools in an appropriate manner. (CLR X)
6. Identify appropriate strategies for solving a problem (CLR XI)

Worth
4.0%
of your total mark

Assignment 3

Room Booking Scheduler, Part 3

Program Description

In this assignment you'll add more features to your RoomBooking Scheduler. For those students who failed to submit Assignment 2 you may use my copy, available on Brightspace, as the starting point for your Assignment 3 code. But be certain to include in your Javadocs at the top of each class: *`@author: yourname, based on code supplied by Prof. Dave Houtman`*

I. Load the Assignment2 project and copy your existing classes to the new Project

a) Create a new CST8284_20W_Assignment3 project folder, copy the `roomScheduler` and `room` packages over from Assignment2 (including all its classes), and refactor the names of the two packages to reflect the change from `asgmt2` to `asgmt3`.

b) There is no UML diagram for this assignment; you are free to implement the requirements any way you wish, so long as it's within the existing UML framework established in Assignment 2 (except where specifically indicated in this document, for example in the next section).

Note however that the principles of good code design still apply. So, at the marker's discretion, you will lose marks for not implementing best practices such as, e.g. not using getters and setters, not chaining methods and constructors, not practicing code reuse, not employing the *principle of least privilege* in your code design, sloppy design that uses several lines of code where only a few are needed, etc.

II. Add the following new features to your Room Scheduler Application

a) Throw a BadRoomBookingException to catch bad data input errors

Add a BadRoomBookingException class to your project, which should extend from RuntimeException. Add two constructors to your new class: a no-arg constructor and a two-String constructor, which passes both a `header` String along with a more detailed `message` String describing the exception and the action the user should take. Pass the message String to the superclass using `super()` (as you did in Lab 7) so that it will be returned when you call `getMessage()`. Store the `header` parameter in a private String field named `header` (which you'll need to add to your new exception class, along with appropriate getters and setters). Chain the no-arg `BadRoomBookingException` constructor to the 2-String constructor, passing the default message "Please try again" and the default header "Bad room rooking entered".

Handling Exceptions Correctly

The great advantage to using an exception handler is that the place where an error occurs can be far removed from where it is caught and dealt with. Logically, you'll want to test the validity of any new piece of data in the appropriate setter to prevent erroneous data from being stored. So you should test the data (using an appropriate method, as in Lab 7) in the setter designed to store that particular piece of data, and throw a `BadRoomBookingException` (with the appropriate header and message) if the data valued is incorrect. But the actual try/catch block should be elsewhere, at a location where the data can be reasonably re-entered without otherwise disrupting the program flow. As always, there is a trade-off between code-readability and catching an error close to the place where it is generated.

A reminder that the purpose of exception handling *is to deal with abnormal execution in an appropriate fashion*, i.e. to avoid dumping the stack trace into the console at all costs. So at no point should you be calling `printStackTrace()`, or throwing exceptions up to `main()` (where they get dumped out onto the console—exactly what we don't want), this despite the fact that these 'solutions' frequently appear in the 'Quick Fix' pop-up that offers suggestions on how to 'resolve' problems.

Example of Bad Input	Header String	Message String
	Bad Calendar value input	

`Start Time: 4 p.m.`
`End Time: 15:00`
	"End time precedes start time"	"The room booking start time must occur before the end time of the room booking."

`Start Time: 4 p.m.`
`End Time: 16:00`
	"Zero time room booking"	"Start and end time of the room booking are the same. The minimum time for a room booking is one hour."

`31042019`
`29022101`
`20132019`
`22-13-2019`
`20190312`
`20/03/2019`
	"Bad Calendar format"	"Bad calendar date was entered. The correct format is DDMMYYYY."

	Bad first or last name *(i.e. an exception is thrown if any of the following errors occur in the first or last name)*	
`R*bert J6son`	"Name contains illegal characters"	"A name cannot include characters other than alphabetic characters, the dash (-), the period (.), and the apostrophe (')."
`Dooowhaadiddydiddy dummdiddydoooo Dada`	"Name length exceeded"	"The first or last name exceeds the 30 character maximum allowed."
	Bad phone number	
613-555-121 (613) 555-1212 6135551212 613-55A-1212	"Bad telephone number"	"The telephone number must be a 10-digit number, separated by '-' in the form, e.g. 613-555-1212."
	Bad general input	
(Empty value entered)	"Missing value"	"Missing an input value."
(`null` value entered)	"Null value entered"	"An attempt was made to pass a null value to a variable."

In your existing code, throw a `BadRoomBookingException` for each of the exceptions indicated in the table above. The header to be displayed along with its associated message is indicated in the table; you can use your new `getHeader()` method, along with the `getMessage()` superclass method whenever you need to output this information.

As with Lab 7, you should write a special method for each of the exceptions listed above to test the validity of the data.

The marker of your assignment will be running a series of tests on your code to check that each type of `BadRoomBookingException` is thrown correctly for each error condition indicated. So test your code thoroughly to make certain that the appropriate exception is generated for each of the error conditions listed.

In particular, note:

(1) Bad phone numbers may be tested for by using regular expressions ('regex'), if you are familiar that topic. But for most students,

using some combination of the standard String methods (such as `contains()`, `split()`, and `length()`) in conjunction with a method such as `Integer.decode()` will form a much quicker and more reliable test than attempting to figure out regex;

(2) Strictly speaking, telephone numbers cannot begin with a 0 or 1; we'll ignore that rule in this assignment. Use the samples shown in the left column as your guide to what constitutes a valid phone number;

(3) An empty string is not the same thing as a null string. The former occurs when a string is equal to "", the latter when the string is `null`. The latter would trigger a `NullPointerException` if a method was called on a `null` string; the former will not. So one way to test for a `null` is to deliberately throw a `NullPointerException` in your code during testing—*which must be removed from your finished submission.*

Note that *all* inputs, including those associated with the exceptions listed in the table above must test for empty or null input;

(4) You can catch one exception—such as a `NullPointerException` or a `NumberFormatException`—and use it to throw another, such as a `BadRoomBookingException`;

(5) As always, you are required to document any solutions you find on the internet. However, since you're expected to derive the exception-detecting algorithms yourself (for example, the test for a valid phone number), you will not get any marks for solutions which are not your own. This applies to the other requirements in this document as well.

b) **Refactor your code to sort and search the RoomBookings ArrayList using the Comparator interface**

Recall that in Assignment 2 we searched through our ArrayList of RoomBookings by laboriously looping through each element until we found the Calendar object we were looking for. Clearly, there must be a better way to do this.

The 'better way' is to use the `Collections` class's `sort()` and `binarySearch()` methods. Then, when we wish to locate or display a RoomBooking, the process will be faster and more efficient. To utilize these methods, we first need to implement an appropriate `Comparable` or `Comparator` class, so begin by creating a new class `SortRoomBookingsByCalendar` that implements either of these interfaces. Inside that class, you'll need to override the `compareTo()` or `compare()` method so that it returns an integer value representing the difference between two RoomBooking's Calendars. Then, use an instance of your `SortRoomBookingsByCalendar` class to sort the ArrayList using the `Collections sort()` method. For details on this, see the course notes, or consult the web for guidance (as always, be sure to cite your sources). The following will help get you started:

https://www.baeldung.com/java-comparator-comparable

https://www.geeksforgeeks.org/comparator-interface-java/

https://howtodoinjava.com/sort/sort-arraylist-objects-comparable-comparator/

Once the ArrayList of `RoomBookings` is sorted, searching for a particular RoomBooking by Calendar date is straightforward, this time using the `Collections.binarySearch()` method. Again, you'll probably want to do some research on this. The following websites should help get you started:

https://www.geeksforgeeks.org/collections-binarysearch-java-examples/

https://www.concretepage.com/java/example-collections-binarysearch-java

Use the `Collections sort()` and `binarySearch()` methods in your existing code to refactor *both* the `findBooking()` and `displayDayBookings()` methods. (The latter can use `binarySearch()` on a sorted ArrayList of `RoomBookings` to find the first occurrence of a `Calendar` date, then loop through the following `RoomBooking`s until a new Calendar date is encountered.) Using the 'sort and search' strategy listed above, we only need to sort once, search for the first matching Calendar day, then step through the ArrayList from that point onward until the current day is done—a far more efficient strategy than checking for each RoomBookings in the entire ArrayList to check the Calendar for each hour on a particular day.

You may find that some of your other methods need refactoring as well once you make the above changes, depending on how you implemented your code. At very least, you must refactor `findBooking()` and `displayDayBookings()` using the `Collection` methods specified to fulfill the requirements for this section of Assignment 3.

Finally, note that if each time you add a RoomBooking to the ArrayList (in saveRoomBooking()), you sort the ArrayList as well, you can save yourself some time whenever you need to search the ArrayList.

c) Document your code using JavaDoc

JavaDoc adds hypertext-linked documentation to your code—you saw it briefly in Lab 1— allowing you to output rich text comments similar to those seen on the Oracle web site for each Java class. Provided, that is, you take the time to write and lay out your comments correctly. (See the sample documentation, downloaded from the Oracle web site, below.) A good place to begin with JavaDoc is

https://idratherbewriting.com/java-javadoc-tags/

A doc comment is written in HTML and must precede a class, field, constructor or method declaration. It is made up of two parts -- a description followed by block tags. In the example below, the block tags are @param, @return, and @see.

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
 public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
 }
```

From: "How to write Doc Comments for the JavaDoc Tool", Oracle Corp., downloaded from http://www.oracle.com/technetwork/articles/java/index-137868.html *April 11, 2017* which explains what each tag does. For this assignment, the following tags are essential:

For each class:	For methods/constructors
@author	@param
@version	@returns
	@throws

(You can safely ignore @see and @since for this assignment.) Feel free to implement any other JavaDoc and HTML tags you feel are appropriate. Also, you'll need to refer to the *Algonquin College Documentation Standard* (ACDS), a set of (somewhat dated) slides indicating which components of your code need to be documented (short answer: pretty much everything). This document is available on Brightspace. The ACDS specifies that you must document both the file *and* the class (because a file may contain more than one class.) In this assignment, you're fee to ignore the first requirement: *document each class only*.

Your efforts to document your code correctly will all be for nothing if you don't follow the instructions in the Appendix below *exactly*. This is because the JavaDoc utility does not operate on all the folders in your project automatically, but *only* on the Project Explorer item last selected at the time you ran the utility. So if you click on a *class* and then execute the JavaDoc utility, you will only generate JavaDoc for the methods and constructors *in that class*—and nothing else. And you will get '0' for your documentation as a result, regardless of how detailed the description of your code was inside *each* class, since *you are marked on your JavaDoc output only, not on the comments in your Java code.*

Lastly, note that JavaDoc replaces the documentation you were required to provide at the top of each class in Assignments 1 and 2. So remove these comments, as they will now be superseded by the more extensive JavaDoc comments you'll provide. And while you're tidying up your documentation, remove any comments or commented-out code that is not part of your submission: if it's not part of the code or documentation, I don't need to see it.

III. Notes, Suggestions, Tips, and Warnings

a) Your documentation needs to be complete. This means: (1) every method is documented, even one-line getters and setters; and (2) you must give complete descriptions in your documentation. For example, trivial documentation such as "Getter for telephone number" will cost you marks, since it doesn't say anything that isn't obvious. Does the setter throw an exception? Is it only called by the constructor? Can it be set to null, or empty, and if so under what circumstances? Proper documentation includes an analysis of all these things, things which "Getter for telephone number" doesn't begin to address.

   Use the Oracle Java documentation for any class as your guide to what good documentation should look like.

b) Students are reminded that:
   - You should not need to use code/concepts that lie outside of the ideas presented in the course notes;
   - You *must* cite all sources used in the production of you code according to the format provided in Module00. Failure to do so *will* result in a charge of plagiarism. The one exception is the information provided in the course notes themselves;
   - Students must be able to explain the execution of their code. If you can't explain its execution, then it is reasonable to question whether you actually wrote the code or not. Partial marks, including a mark of zero and a charge of plagiarism, may be awarded if a student is unable to explain the execution of code that he/she presumably authored.

IV. Submission Guidelines

Your code should be uploaded to Brightspace (via the link posted) in a single zip file obtained by:

1) In Eclipse, right click on the ***project*** name (`CST8284_20W_Assignment3`);
2) select 'Export' then select General >> Archive File, then click Next;
3) in the Archive File menu make sure *all* of the project subfolders are selected (`.settings`, `src`, `bin`, and especially `doc`) and the 'Save in zip format' and 'Create directory structure for files' radio buttons are selected;
4) In the 'To Archive File' window, save your zip file to a location you'll remember. But make certain to name your zip file according to the following format, as outlined in Module 00:

Assignment3_Yourlastname_Yourfirstname.zip

including the underscores and capitals, but with *your* last and first name inserted as indicated. Failure to label your zip file correctly will result in lost marks.

Note:

- A good safety precaution is to always take the .zip file you've just uploaded to Brightspace and open it on your laptop. Better still, in Eclipse, import the zipped file and check to make sure everything is there;
- You can upload as many attempts at Assignment 3 as you'd like, but only the final attempt is marked

Corrections and Addenda:

Version 1.01 (March 19): You *can* implement your code to catch data entry errors as soon as they are entered; or you may wish to wait until the user has entered all the data, and only then prompt for re-entry of any incorrect data. Since programmers prefer to store all the data at once using an appropriate constructor (and the constructor will use setters, which will be throwing the exceptions), the latter choice leads to tidier code. But the former choice, instantiating a no-arg constructor, setting each field separately, and catching the error promptly, makes for a better user experience. So either way works, and is fine for this assignment.

But keep in mind that in Assignment 4 you'll be converting your program to a GUI interface. Then, you'll enter everything into a dialog window first and then click a button to store the information all at once (rather like hitting the 'Submit' button to enter data into a web form.) You may want to consider this fact when building your Assignment 3.

Assignment 3 Marking Guide

Requirement	Mark
File labelled and zipped correctly, as per instructions above. The doc folder is supplied in your zip file and displays all the classes used in your project, hypertext-linked as required	**3**
Exception handling has been added to the code so that the user is protected from storing bad dates, times, phone numbers or names in the RoomBooking ArrayList. Additionally, the process of re-entering data should be easy for the user; it should not involve awkward work-arounds. All inputs should be checked for null and empty inputs.	**9**
`findBookings()` and `displayDayBookings()` have been refactored using `Collections sort()` and `binarySearch()` methods as indicated.	**6**
All classes, fields, methods and constructors have *complete* JavaDoc comments according to ACDS.	**6**
MINUS: late penalty; failure to cite sources; private information not kept secure through data hiding; diagnostic strings output to the console, abnormal termination, exceptions thrown under certain circumstances; unusual, abnormal and erratic features displayed during execution; your documentation should not include comments which are not part of the program itself (e.g. TODOs and commented-out code: these are for your purposes only: *if it's not part of the program, I don't need to see it).*	
TOTAL :	**24**

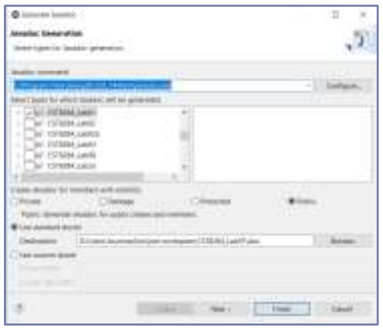
Appendix: Generating JavaDocs

a. To generate JavaDoc, right click the *project* from the Package Explorer. (This is an essential step, and the contents of your Javadoc folder will be incorrect if you do not follow this simple instruction.) From the menu along the top of the Eclipse screen, select Project >> Generate Javadoc…

b. In the dialogue that appears, make sure the current assignment is selected from the list of types for which Javadocs will be generated.

c. On occasion, Javadoc needs to be configured. If the Finish button is grayed out, then select the 'Configure…' button, and navigate to the folder in which the JDK is located. The javadoc.exe file is located in Java's `bin` directory. Hence your Javadoc path should look something like: C:\Program Files\Java\ jdk1.8.0_*version*\bin\javadoc.exe

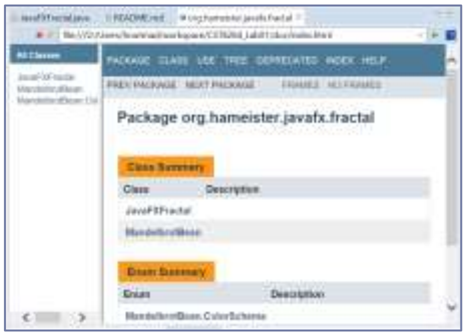
If it doesn't, you may need to navigate to this file and install it manually in Eclipse.



d. Select the Finish button. If you get additional messages during Javadoc setup, select the defaults. (Note that if you have any other tabs open in Eclipse, including malfunctioning code from previous projects, the Javadoc utility may get confused. Close the tabs for any old files that you've been working on, those which don't have anything to do with your current project. Then run the Javadoc utility again, being careful to select the project file before starting.)

e. Inspect the doc folder which appears in the CST8284_20W_Assignment3 project: this is where your Javadoc files are located. (Note that the doc folder must be in your final submission, or you JavaDoc mark will be 0, even if you Java code is documented.) If there were errors in your Javadoc tags, javadoc may halt execution. You'll need to fix these errors to generate the `doc` folder required in the output you submit.

f. Most of the files in the `doc` folder are HTML files, which can be opened in Eclipse simply by double-clicking on them. Double click the index.html file (this is the main point of entry for Javadoc files) and a web page, like the one shown below, should appear in the Eclipse code window. Javadoc files are hypertext linked to one another, hence you can click on any link in any Javadoc HTML file to take you to any other file.


PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package org.hameister.javafx.fractal

Class Summary

Class	Description
JavaFXFractal	
MandelbrotBean	

Enum Summary

Enum	Description
MandelbrotBean.ColorScheme	

If you see only HTML code then you should right click on index.html and select Open With > Web Browser, which should then display the web page rather than the HTML text file on which it is based. But if your Eclipse setup is configured correctly, you should see the web page in the Eclipse code window, like the one shown above.