Version Number:
1.11 (Jan. 30, 2020)

Assignment 1

Room Booking Scheduler

To be submitted online not later than Sunday, February 9th, 2020, 11:59 p.m.

Description:

The purpose of this assignment is to build a room booking scheduler. This basic program, involving arrays of objects, chaining, overloading, and getters/setters, will further evolve as we add new features in Assignments 2, 3, and 4. In building this application, you will demonstrate your understanding of the following course learning requirements (CLRs), as stated in the *CST8284—Object Oriented Programming (Java)* course outline:

1. Install and use Eclipse IDE to debug your code (CLR I)
2. Write java programming code to solve a problem, based on the problem context, including UML diagrams, using object-oriented techniques (CLR II)
3. Use basic data structures (CLR III), including implementing arrays of primitive and reference types
4. Implement program Input/Output operations (CLR V)
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
7.5%
of your total mark

Assignment 1

Room Booking Scheduler

Program Description

This application will allow the user to store and retrieve booking information from a room scheduler. The scheduler is (at present) very general, and could potentially be used to book any kind of room, e.g. a ballroom, a computer lab, an executive meeting, a conference hall, etc..

Regardless of the type of room or the nature of the meeting, each booking will need to contain the following general information:

1. the contact information of the person that booked the room, including their name and (optionally) organization, along with their telephone number;
2. the date, and starting and ending time the room is booked for;
3. information about the activity involved (e.g. meeting type, security access, event number, event catered or not, etc.).

In this first assignment, a `RoomScheduler` object contains an array of `RoomBooking` objects, and each `RoomBooking` object stores the three pieces of information indicated above. (Since our initial application assumes that only one room will be booked, we'll ignore the room and its details for now.)

To simplify matters somewhat, we'll assume each room can only be reserved for intervals that start on the hour, beginning at 8 a.m. up to 10 p.m.; that is, the last possible booking of any day runs from 22:00:00 p.m. to 22:59:59, for a total of (potentially) 15 room bookings each day, (assuming each was of one-hour duration). However, our `RoomScheduler` application will allow the client to book the room for the whole day (if needed), from 8 a.m. to 11 p.m., or for any fraction of the day as well.

Any particular `RoomBooking` may be located by looping through the array of `RoomBookings`, searching each object by its date/time/hour, until the last loaded element of the array is reached. This method can then be used successively to display the schedule of all bookings for a particular room on a particular day.

In the present implementation of this program, `RoomBookings` cannot be deleted (yet), nor can the array be sorted (yet); these and other features will be added in subsequent assignments.

I. Create the Project, package and classes

a. In Eclipse, create a new project called `CST8284_20W_Assignment1`. To this project add a package called `cst8284.asgmt1.roomScheduler`. Then, following the UML diagram below, add each of the six classes indicated to the package.

b. Before you proceed to the details of each class in Section II, note the following rules about this assignment, which must be strictly adhered to:

1. Follow the UML diagram *exactly as it is written*, according to the most up-to-date version of this document. You cannot add members that are not written in the UML diagram, nor can you neglect any of them either—and this applies to features that don't appear to have any use in your code (at present; on this note, be sure to read item (2) below). *If it's written in the UML, it's not optional: write the code you're directed to write, or marks will be deducted.*

   And if you want to get creative with the requirements, you may. *But do so on you own time, and do not submit your 'improved' version and expect to get marks for it.* In short: if you hand in code that differs in any way from the UML, you will lose marks. Of course, how you choose to write each component of the program is

up to you. But the way in which the pieces connect is directed by the UML.

2. Aside from a few getters and setters used in the smaller classes, all methods indicated in the UML are expected to be put to use in your code. So if there's a method that's required in the UML and you're not using it, you're *likely* missing something, and your code is not being used correctly—and you will lose marks. Take the UML as an indication of *not just of what needs to be coded, but of how the code is to be connected in a well-written, optimized application* (given the rather limited requirements of this assignment).

   So if you can't figure out how certain features (such as the `private static final` constants declared in the `RoomScheduler` class) will be used in code, think first, then ask if you're stuck: these things exist for a reason.

3. Employ best practices at all times: reuse code, don't copy and paste; don't declare variables unnecessarily; keep your code to the minimum needed to ensure the program functions reliably; only setters and getters talk directly to private fields, everything else talks to the public setters and getters; and chain all overloaded constructors and methods. This applies, e.g. between the `toString()` method used by the `RoomBooking` class and the `toString()` methods used by its composite members, such as `ContactInfo` and `Activity`.

Additional information on the expectations for this project are included at the end of this document in Section III, 'Notes, Suggestions, Tips, and Warnings.'

II. Write the code indicated for each class

As seen in the UML diagram below, six classes will be needed in this application: `RoomSchedulerLauncher`, `RoomScheduler`, `RoomBooking`, `TimeBlock`, `ContactInfo`, and `Activity`. Additionally, the `TimeBlock` class makes heavy use of Oracle's `Calendar` class, which, in addition to storing the date, holds the start and end times of each `RoomBooking`'s `TimeBlock`. Information on `Calendar` is available at several sites, but the following contains useful examples for those unfamiliar with this class; it's probably the best way to get started with `Calendar`. See: https://www.geeksforgeeks.org/calendar-class-in-java-with-examples/. As always, be sure to cite any websites you used during the construction of your application.

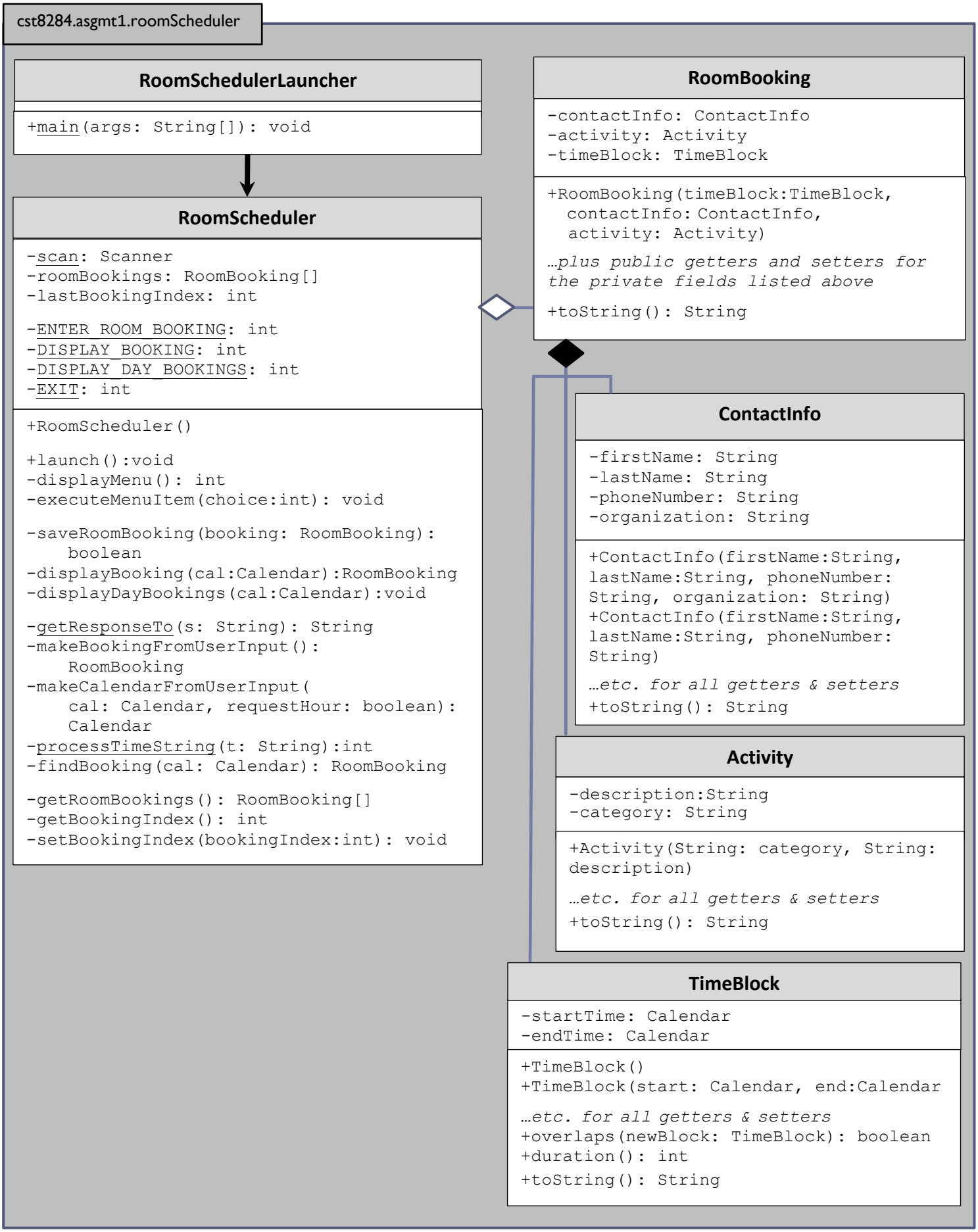
To instantiate a new Calendar object, you use

```
Calendar cal = Calendar.getInstance();
```

Here, `getInstance()` is an example of a *factory method*, a method that produces a Calendar object for you. You must use this method to instantiate a new Calendar object rather than call `new Calendar`. Each time you call this method the Calendar created will record the moment of its creation in its various fields, including the year, month, day, hour, minute, second and millisecond. If two Calendar's are created even a fraction of a second apart, then their milliseconds values will be different. So even if two Calendar's have the same year, month, day, and hour, minute and second, the Calendar `equals()` method to compare two Calendars will fail, because their milliseconds values are different. To ensure that all fields are cleared after a new Calendar is created (but before you set its year, month, day and hour) be sure to use

```
Calendar.clear()
```

Also note (1) months are 0 – based in Calendar; thus Jan is month 0, Feb is 1, etc. Since this isn't the way humans think about months, you'll need to offset the user input by 1 to satisfy Calendar's requirements; (2) Calendar.HOUR only return values 0 to 12 (i.e. it uses a.m. and p.m.); Calendar.HOUR_OF_DAY returns 24-hour values. So be sure to use HOUR_OF_DAY in your code, or your methods won't work correctly; (3) To view the Calendar's hours field in Debug, look in the `fields` array, element `[11]`


cst8284.asgmt1.roomScheduler
RoomSchedulerLauncher
+main(args: String[]): void
RoomScheduler
-scan: Scanner
-roomBookings: RoomBooking[]
-lastBookingIndex: int
-ENTER_ROOM_BOOKING: int
-DISPLAY_BOOKING: int
-DISPLAY_DAY_BOOKINGS: int
-EXIT: int
+RoomScheduler()
+launch():void
-displayMenu(): int
-executeMenuItem(choice:int): void
-saveRoomBooking(booking: RoomBooking): boolean
-displayBooking(cal:Calendar):RoomBooking
-displayDayBookings(cal:Calendar):void
-getResponseTo(s: String): String
-makeBookingFromUserInput(): RoomBooking
-makeCalendarFromUserInput(cal: Calendar, requestHour: boolean): Calendar
-processTimeString(t: String):int
-findBooking(cal: Calendar): RoomBooking
-getRoomBookings(): RoomBooking[]
-getBookingIndex(): int
-setBookingIndex(bookingIndex:int): void
RoomBooking
-contactInfo: ContactInfo
-activity: Activity
-timeBlock: TimeBlock
+RoomBooking(timeBlock:TimeBlock, contactInfo: ContactInfo, activity: Activity)
…plus public getters and setters for the private fields listed above
+toString(): String
ContactInfo
-firstName: String
-lastName: String
-phoneNumber: String
-organization: String
+ContactInfo(firstName:String, lastName:String, phoneNumber: String, organization: String)
+ContactInfo(firstName:String, lastName:String, phoneNumber: String)
…etc. for all getters & setters
+toString(): String
Activity
-description:String
-category: String
+Activity(String: category, String: description)
…etc. for all getters & setters
+toString(): String
TimeBlock
-startTime: Calendar
-endTime: Calendar
+TimeBlock()
+TimeBlock(start: Calendar, end:Calendar
…etc. for all getters & setters
+overlaps(newBlock: TimeBlock): boolean
+duration(): int
+toString(): String

Regarding the six classes you'll be coding: there's a few general ideas to keep in mind as you proceed (specific details on each class follow below). Most of what you need to know *should* be obvious from the UML diagram, but the following three items may be somewhat subtle, and can only be inferred by careful examination:

First: Only the `RoomScheduler` class handles the input and output of information. You should not have, nor need to use, Scanner *in any other class*. And the same goes for inputting Strings: only `RoomScheduler` handles I/O. Other classes may *store* String (and other) data, but all access to these Strings is made via the appropriate public methods of that class, such as `toString()`, and that class's getters and setters.

Second: We'll assume, for now, that the user of our application is well-behaved and will not deliberately be entering any bad data—an assumption we'll overturn in a later assignment. But at this point, there's no need to check the validity of the data.

Third: remember that private properties are never used directly; everything goes through getters and setters. For example, the statement:

```
getRoomBookings[getLastBookingInde
x()].getContactInfo().toString()
```

returns a String representing the `Contactinfo` of the `RoomBooking` object stored in the `roomBookings` array having the index value of the last booking in the array. Whether you understand everything in this statement or not doesn't matter at the moment; what matters is that at no point does the code access a private property directly: *<u>everything</u> is done through getters and setters*.

A detailed description of each of the above six classes used in this application follows:

a) **RoomSchedulerLauncher** is the main point of entry for this program, hence it will contain the `main()` method. It has only one purpose, which is to launch the application by first instantiating a new `RoomScheduler` object, and then calling its `launch()` method—that's all.

b) Each instance of a **RoomBooking** object is characterized by the three properties indicated at the start of this document: the `ContactInfo`, which saves the information on who booked the room; an `Activity`, which is used to store, at minimum, a description of what the booking is about; and a `TimeBlock`, that stores the starting and ending times of the event the room is booked for along with the date, in `Calendar` form. (Note that the start and end time should always have the same day, month and year; a `RoomBooking` is never for more than a day at a time.)

As indicated in the UML diagram for `RoomBooking`, you must write getters and setters for each of these three fields. And note that the `RoomBooking` `toString()` method will need to call upon the `toString()`'s of its three composite classes. (See the output for an indication of which class outputs what information.) So constructing the `RoomBooking` class itself should be fairly straightforward.

c) The **ContactInfo** and **Activity** classes are similarly obvious, given the UML diagram provided. The latter class takes two strings, a one- or two-word `category` for the event (like 'Wedding", "Bar Mitzvah" or "Board Meeting") and a more general `description` to describe the event; the former class takes the first name, last name, and phone number of the contact, and (optionally) organization name. Note that all these values are passed as Strings to that class's setters in `RoomScheduler`. Note also that `ContactInfo` has an overloaded constructor, which you'll need to chain. For `ContactInfo`'s three-arg constructor, assume the default organization is

"Algonquin College". (See the Addenda at the end of this document for clarification.)

d) The **TimeBlock** class is slightly busier. As previously indicated, it stores both the start time and end time of any booked event as `Calendar` objects—which use the 24-hour clock, with 0 as midnight, and 12 as noon. But each `Calendar` object stores other information aside from the hour of the day, including the day, month, year, minute, second and millisecond—much more than you'll need for booking a room that stores each meeting on the hour. (Of course, we also care about the day, month, and year of an event as well, but these are fairly easy to deal with; the time and duration of the booking make it more problematic, as you'll see shortly.)

   To simplify the output somewhat, it's easier to use `Calendar`'s `getTime()` method whenever you need to display `Calendar` output. `getTime()` returns a `Date` object, and while many of `Date`'s methods have been deprecated, it still finds wide use. In particular, you'll find `Date`'s `toString()` output much easier to process than `Calendar`'s `toString()`. So it's highly advisable to use a `Date`'s `toString()` method for your output.

   Note that the no-arg `TimeBlock` constructor should instantiate a `RoomBooking starttime` from 8 a.m. to 10 p.m. (i.e. `endTime` = 23) on the day the program is executed (which is the default for a `Calendar` object whenever a new `Calendar` is instantiated).

   Finally, note the two service methods in the `TimeBlock` class. `duration()` returns the span of hours the room booking is for. Thus if `startTime` is 9 and `endTime` is 17, the duration of the booking is for 8 hours.

   `overlaps()` allows the user to query whether a new `TimeBlock` will conflict with *this* `TimeBlock`. That is, does the `newBlock` end after an existing block starts? Then a booking conflict would occur, and the method should return `true`. Similarly, if the user attempted to book a `newBlock` of time that started before an existing booking was finished, this too should return `true`.

e) Most of the real action—and most of the work you'll need to do to make this application function reliably—is done in **RoomScheduler,** which acts like the control centre for the program: it handles *all* I/O requests, as well as loading, finding, and displaying the `RoomBooking` objects stored in the `roomBookings` array. We can categorize `RoomScheduler`'s methods according to their role in the program as follows:

   Initialization/Menu Methods

   The public **`launch()`** method calls two methods, one for displaying the menu to the user and returning their choice (**`displayMenu()`**) and the second for executing the menu choice selected (**`executeMenuItem()`**). This makes testing easier, and has the added benefit of allowing for a smoother transition to a GUI interface in Assignment 4. As indicated in the sample output below, this code should loop *inside* `launch()`, calling these two methods in succession until the **`EXIT`** value is selected to quit the application.

   Note the four fixed constants, **`ENTER_ROOM_BOOKING`**, **`DISPLAY_BOOKING`**, **`DISPLAY_DAY_BOOKINGS`**, and **`EXIT`**.

   These *must* be used in your code wherever the integer values they represent would normally appear. This eases code readability, particularly in the `switch()` statement that determines which action to execute based on the menu presented to the user. Your code should be written in such a way that if the numerical value attached to any constant was changed (say by selecting '10' to exit, or even 'X' to exit), your code

would not break, i.e. your prompts to the user would still be correct, without having to rewrite the code in `RoomScheduler`; only the fixed constant value itself would change, *nothing else*.

Note!

(1) You should use switch() *in preference to a series of* if…else if *statements whenever there are more than three choices to be made. A long series of* if…else if *statements is somewhat unwieldy;* switch()…case *is a more elegant solution in this situation. Therefore, I expect to see* switch() *used correctly in executeMenuItem();*

(2) Under no circumstances should your code call itself recursively. A do…while *loop is the correct way to implement your launch() method;*

(3) Under no circumstances should you terminate your program using System.exit(). When the user selects EXIT, simply fall through the loop, and exit the program normally.

Array control methods

First, a comment on the `roomBookings` array. This will store `RoomBooking` objects loaded sequentially in the order in which they were created; there is no attempt to sort, delete, or consolidate the contents of the array at this stage. As always, never access `roomBookings` directly; use **`getRoomBookings()`** to access this array.

To find a particular `RoomBooking`, step through the array up to the value returned by **`getBookingIndex()`** (which returns the location where the last `RoomBooking` object was loaded in the array), searching the **`TimeBlock`**s as you go for a match to the desired time/date. More on this shortly.

Remember to increment the `lastBookingIndex` after a new `RoomBooking` object has been added to the array (using **`setBookingIndex()`**), or you'll overwrite the existing array element when you intended to add to the next. The **`saveRoomBooking()`** method attempts to save a booking into the `roomBookings` array. But before it can do this, you'll need to (1) check that the `RoomBooking` passed to the method is not `null`, and (2) make sure the `RoomBooking`'s `TimeBlock` does not conflict with an existing `RoomBooking` already in the array (since just because our putative user isn't willfully malicious doesn't mean he isn't negligent). The appropriate way to do this, given the methods available, is to execute the `findBooking()` method, discussed below. If it returns `false`, there is no conflict, and the new booking can be loaded into the array at the location pointed to by `getBookingIndex()`. Once safely booked, `saveRoomBooking()` returns `true` to signify success; otherwise `false` if there was a problem. The boolean value returned can then be used to print out an appropriate message, depending on whether the booking was successful or not.

Convenience methods

The `RoomScheduler` class contains five *convenience methods*, that is, methods that act as building blocks for other methods. Convenience methods often form the 'workhorses' of any application, doing common tasks that help simplify the rest of the coding. `RoomScheduler`'s five convenience methods include:

`getResponseTo()`. This is a simple method that prints out the String entered as a parameter, and then scans in, and returns, the user's response as another String. This method saves us having to repeat this operation every time we wish to get input from the user. It's easy: here's the code, which you should enter into the `RoomScheduler` class:

```
private static String
        getResponseTo(String s) {
  System.out.print(s);
  return(scan.nextLine());
}
```

(where `scan` the Scanner object indicated in the UML diagram).

`makeCalendarFromUserInput()` does just that—it prompts the user for the date and time of a booking by calling `getResponseTo()` twice, once for a day/month/year and a second time for the hour. It processes these two Strings into year, month, day (entered as a String in the format DDMMYYYY) and hour values, and then instantiates a new `Calendar` object using `Calendar`'s variously overloaded `set()` methods. The method returns a loaded `Calendar` object, which can then be used to set, find and display `RoomBooking` objects from the `roomBookings` array.

Note that `makeCalendarFromUserInput()` takes two parameters: an existing `Calendar` object, and a boolean value called `requestHour`. The former is used if the user has already entered a starting date and starting time for a booking; no need to ask them to re-enter the date, since all bookings are assumed to have the same day, month, and year. So when `Calendar` is not `null`, the `Calendar` returned will have the same day, month and year as the `Calendar` parameter passed. But if the first parameter is `null`, the user will need to be prompted to enter the date in DDMMYYY format.

The second parameter determines whether the user needs to be prompted to enter the hour or not. This is useful when all we need is the date of the Calendar (say, to find all the bookings on a particular day) and don't care about the time of a booking. So when this parameter is false, your code should suppress the urge to prompt the user to enter a time.

When the user does enter a time, it could be in either a.m.-p.m. format, as follows:

```
8 a.m.
9 am
1 p.m.
```

or in 24-hour time format, like this

```
8:00
13:00
23:00
```

Either of these two formats, entered as a String, would be considered valid. The **`processTimeString()`** method is responsible for taking the time String (as returned by `getResponseTo()` whenever `requestHour` is `true`), and returning an integer value, corresponding to the 24-hour format expected by the `Calendar`'s `set()` method.

In `processTimeString()` you may use any method you choose to convert the String to an int, however *you must NOT use a regular expression ('regex') at this stage.*. (Furthermore, you should not use `charAt()` to read the string in one character at a time.) You should be able to parse the input time String using just the standard String class methods alone—such as `split()`, `substring()`, `trim()`, `replace()`, and `contains()`. As always, cite all sources you employ in writing your code.

`makeBookingFromUserInput()` does the much the same thing as `makeCalendarFromUserInput()`, but prompts the user for *all* the RoomBooking information—the client's first and last name, phone number, and organization (used to instantiate a new `ContactInfo` object); the category and description Strings (used to instantiate a new `Activity` object); and the date and time of the booking, using `makeCalendarFromUserInput()`—called twice—to generate the `startTime` and `endTime` needed to instantiate a new `TimeBlock`. And each of these rely on `getResponseTo()` to prompt and return from the user the Strings containing this information.

Finally, **`findBooking()`** returns a `RoomBooking` object based on a `Calendar` input (including, of course both the date and time). Your code will need to (1) instantiate

a new, 1-hour TimeBlock, based on the Calendar object entered, (2) loop through the roomBookings array and check each RoomBooking for an overlap by comparing it with the new **TimeBlock**. If the overlaps() method returns true, quit looping: you've found the desired RoomBooking, which then gets returned by the method. If there is no overlap with the Calendar's date/1-hour block of time is found by the time you get to the lastBookingIndex, return a null.

Note that if an array element is null (i.e. there is no RoomBooking object stored at a particular location in the array), then you cannot call its overlaps() method, since null.overlaps() triggers a NullPointerException. So be sure to test that each roomBookings element is not null before calling that element's overlaps() method. If that element is null, skip the test—there's nothing stored in the array at that index—and move on to the next element of the array.

Menu Method Calls

The final category of RoomScheduler methods includes those methods associated with the three main menu items:

saveRoomBooking, displayBooking(), and displayDaysBookings().

saveRoomBooking() is called whenever the user selects ENTER_ROOM_BOOKING. It takes, as its argument, a RoomBooking, so you'll need to call makeBookingFromUserInput() first to prompt the user to enter new RoomBooking information; this gets passed to saveRoomBooking(). Inside this method, you'll first want to check to see if the booking already exists; use findBooking() to see if any of any matching Calendar objects exist in the roomBookings array. If findBooking() returns null, then the new RoomBooking's TimeBlock is available, and you can safely book the RoomBooking into the array—simply add it to the location pointed to by getRoomBookingIndex(). If not, then abort the attempt, and output a message informing the user that the attempt failed, complete with the conflicting RoomBooking information, which should be returned by the saveRoomBooking() method.

To display that RoomBooking, use the **displayBooking()** method, which is associated with the second menu item. This takes a Calendar object as its parameter. When used with DISPLAY_BOOKING, you'll need to prompt the user to enter a Calendar object first, so call makeCalendarFromUserInput() first, and pass it as the parameter to displayBooking(). (When used with saveRoomBooking() above, use the Calendar associated with the TimeBlock's startTime.) Then pass this Calendar object to displayBooking(). Assuming there is a RoomBooking associated with that Calendar's date and time, display that RoomBooking using that class's toString() method. But if findBooking() returns null, print out the message

"No booking scheduled between X:00 and X+1:00"

where X is the hour of the Calendar object, the time of the booking we'd hoped to display.

Finally, to print out the schedule for an entire day, use **displayDaysBookings()**. To do this, loop through displayBooking(), passing it a Calendar object for each hour of the day, from 8 a.m. to 11 p.m., to test for each potential RoomBooking on that day. (Remember, each Calendar will be converted to a 1-hour TimeBlock in findBooking().) In this case, we need to

initially prompt the user for the date only; it wouldn't make sense to ask for an hour when what we're interested in is the schedule for the entire day. This is where the `boolean requestHour` variable comes in. If you implemented `makeCalendarFromUserInput()` correctly, setting this parameter to false will suppress the prompt for the user to enter the time of the booking. The `Calendar` object returned should have its hour set to 0 initially; you'll reset this value to 8 and start testing from `RoomBookings` on that date between 8 to 22, testing each element in the array to check for a match on that day, for each hour.

In short, displaying a day's booking means testing the elements of the `roomBooking` array for each hour of the day. (This is a most inefficient way to generate a list of `RoomBooking` objects, but we'll be correcting it shortly, in Assignment 2.)

Note that you should use `RoomBooking`'s `endTime` to skip over a block of booked time, e.g. if you find a `RoomBooking` from 8 to 12, you DON'T need to output

"No booking scheduled between 9:00 and 10:00"
"No booking scheduled between 10:00 and 11:00"
"No booking scheduled between 11:00 and 12:00"

—you can skip to 12 o'clock and start testing the `RoomBookings` from noon onward. So whenever you find a booked `TimeBlock`, output its `RoomBooking` information using `toString()`, and then skip to the next unbooked `Calendar` hour.

III. Notes, Suggestions, Tips, and Warnings

a. After just a few hours of working on your program, *you* are the world's expert on its operation. It's unreasonable to expect that anyone else would be able to jump in on short notice and spot your errors (the more so if your code is unstructured and undocumented). Therefore, before requesting assistance from the instructor, you should set breakpoints in your code at the 'last known good' location and step forward from there using Debug until the error is encountered. Be sure to read the error output in the console, to help locate where the problem first occurred. Debug your code at that location. Run the code again until the next error is encountered. Reset the breakpoint to the last 'good' location, and repeat this operation as many times as is required to get the code working. And only then, if you're truly stuck, contact the instructor. But do so only after you've made a valid attempt to fix the program yourself using Debug.

b. You are ***not*** required to supply documentation with this assignment, with the following exception. Each class must include, as the top, the following information:

```
/*  Course Name:
    Student Name:
    Class name:
    Date:
*/
```

with the appropriate information supplied. (The Class here refers to the name of the class itself, not the course number. And the Date is the date on which you wrote the code.)

c. Students are reminded that:
- You should not need to use code/concepts that lie outside of the ideas presented in the course notes (aside from those topics that you are expected to research on your own, like `Calendar`);
- You *must* cite all sources used in the production of you code according to the information provided in Module00; for

your assignments, this includes citing code provided by the instructor. Failure to do so *will* result in a charge of plagiarism. Note that citations should be placed just before the heading of the method in which they are used;

- 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, may be awarded if a student is unable to explain the execution of code that he/she presumably authored. And in situations where the student clearly does not understand the code they presumably wrote, a charge of plagiarism will be brought against the student.

d. The instructor's version of the code will be released on midnight, Feb. 12th, for those students who did not complete this lab on time, or who wish to build their Assignment 2 code on top of the instructor's version (if their own effort was unstable.) Note however, that once the 'official' version is published, it essentially nullifies any outstanding assignments: all unsubmitted Assignment 1's are immediately worth 0. This even applies to students who have been given special dispensation due to personal circumstances; I can only put off releasing my version of the code for a few days after the submission deadline, otherwise a large number of students will not have access to the code they need to start Assignment 2 on time.

e. Sample data is shown at the end of this document. *Your code must be able to run using this data exactly as written (with the possible exception of individual spaces between words and punctuation, which may vary with the implementation)*. A significant number of marks will be removed if the operation of your code diverges significantly from the sample output shown.

A common cause of problems is mixing the use of `nextLine()` with any of Scanner's other '`next()`' methods (like `nextInt()`, `nextDouble()`, and even `next()` itself, to read in Strings.) See the hybrid video on 'The Scanner Bug' and make sure you flush the input stream correctly before prompting the user for a fresh round of input information. Under no circumstances should you instantiate additional Scanner's to solve this problem.

f. You can upload as many attempts at Assignment 1 as you'd like; only the last attempt is marked, everything else is ignored.

IV. Submission Guidelines

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

1) In Eclipse, selecting the ***project*** name (`CST8284_20W_Assignment1`)
2) right clicking on 'Export' and selecting General/Archive File; click Next;
3) in the Archive File menu make sure *all* of the project subfolders are selected (src, bin, .settings) and the 'Save in zip format' and 'Create in Directory Structure' radio buttons are selected
4) In the 'To Archive File' window, save your zip file to a location you'll remember. But make certain the name of your zip file corresponds to the following format, as outlined in Module 00:

Assignment1_Yourlastname_Yourfirstname.zip

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

5) 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, take the zip file you just uploaded, unzip it, and check to make sure everything is there —you'd be surprised how often this simple test fails, and the zip you uploaded turns out to be empty.

Addenda

Version 1.10 (26/01/2020):

Corrections:

(1) Added Sample Output and Marking Scheme to the document

(2) Added information on Calendar in Section III (in the box on Calendar, at the bottom.)

(3) The 24-hour Calendar has the curious feature that there is no 'hour 24'—midnight is 0, not 24. This makes testing for a booking in the last hour of the day somewhat problematic, since your code essentially tests 'if (23 < 0)', when what you want to do is test 'if (23 < 24)'. Rather than convolute the `overlaps()` method further, I've changed the document so that the last `TimeBlock` of any day will end at 11 p.m. rather than midnight. This evades the problem without actually solving it. We may refactor 'midnight' back into the code in a later assignment, but for now, we'll ignore this potential bug through a simple redefinition of the requirements: assume all Calendar hours are less than 24.

(4) A small correction was made to the UML diagram:

descriptionOfWork → description
setBookingIndex() → (corrected)

(5) Change in output String when no booking is found, to conform with sample output (see below)

Clarifications:

(1) When chaining the `TimeBlock` classes, you'll find that the `Calendar` object does not easily allow you to instantiate a `Calendar` object with a specific time. You'll want to use `Calendar.Builder()` instead. To instantiate a `Calendar` object with a set hour (say, 8 a.m.) you'd write:

```
new Calendar.Builder().set
   (Calendar.HOUR, 8).build()
```

You'll probably put `Calendar.Builder()` to use in `RoomScheduler` as well. Note that a builder, like a factory method, is a subject you'll be covering in CST8288 – Design Patterns.

(2) …*CORRECTED BELOW*…Again, a reminder that this constructor must be chained to the second, four-String parameter `ContactInfo` constructor.

(3) There are several ways to convert a String value to an integer, and you may have encountered them already in your CST8110 course. In case you are unfamiliar with this simple conversion, the `Integer.parseInt()` method is probably the fastest, most reliable method to use for this purpose.

(4) The `roomBookings` array needs to be large enough to hold a reasonable number of RoomBookings, so make the size of the array 80 and that will be more than enough for our purposes.

Version 1.11 (30/01/2020)

Corrections

pg. 7, last paragraph: changed to `findBooking()` from `findRoomBooking()` to be consistent with the UML.

pg. 9, line 7: `makeBookingFromUserInput()` in Version 1.10 changed to `makeCalendarFromUserInput()`

pg. 11, second paragraph (2), immediately above: The 3-parameter `ContactInfo()` constructor should chain to the 4-parameter constructor, but with "Algonquin College" used as the default organization. (Note that the original Clarification(2) above mistakenly said 'no-arg' rather than '3-parameter'. it said a few other confusing things, which you should now ignore.)

If the user does not enter any organization—i.e. simply presses ENTER without entering a String when prompted for the organization—then use the 4-parameter constructor as you normally would, but save "" as the fourth parameter. But when it comes time to output the information using `ContactInfo()`'s `toString()` method, rather than print out "Organization: " (followed by nothing, ""), don't print out that line at all: where the user fails to input any organization, output nothing, not even the "Organization: " line.

Sample Output:

```
Enter a selection from the following menu:
1. Enter a room booking
2. Display a booking
3. Display room bookings for the whole day
0. Exit program
1

Enter Client Name (as FirstName LastName): Oly MacDonald
Phone Number (e.g. 613-555-1212): 111-222-3333
Organization (optional): Farmer's Market
Enter event category: AGM
Enter detailed description of event: Annual meeting of farmers
Event Date (entered as DDMMYYYY): 10102020
Start Time: 8:00
End Time: 2 pm
Booking time and date saved.

Enter a selection from the following menu:
1. Enter a room booking
2. Display a booking
3. Display room bookings for the whole day
0. Exit program
2

Event Date (entered as DDMMYYYY): 10102020
Start Time: 14:00
No booking scheduled between 14:00 and 15:00

Enter a selection from the following menu:
1. Enter a room booking
2. Display a booking
3. Display room bookings for the whole day
0. Exit program
2

Event Date (entered as DDMMYYYY): 10102020
Start Time: 11 a.m.
---------------
8:00 - 14:00
Event: AGM
Description: Annual meeting of farmers
Contact Information: Bob MacDonald
Phone: 111-222-3333
Farmer's Market
---------------

Enter a selection from the following menu:
1. Enter a room booking
2. Display a booking
3. Display room bookings for the whole day
0. Exit program
1

Enter Client Name (as FirstName LastName): Mary Meadows
Phone Number (e.g. 613-555-1212): 222-333-4444
Organization (optional):
Enter event category: Wedding Reception
Enter detailed description of event: Reception for Ms. Meadow's daughter and son-in law
Event Date (entered as DDMMYYYY): 10102020
Start Time: 16:00
End Time: 11 pm
Booking time and date saved.

Enter a selection from the following menu:
1. Enter a room booking
2. Display a booking
3. Display room bookings for the whole day
0. Exit program
3

Event Date (entered as DDMMYYYY): 11112020
No booking scheduled between 8:00 and 9:00
No booking scheduled between 9:00 and 10:00
No booking scheduled between 10:00 and 11:00
No booking scheduled between 11:00 and 12:00
No booking scheduled between 12:00 and 13:00
No booking scheduled between 13:00 and 14:00
No booking scheduled between 14:00 and 15:00
No booking scheduled between 15:00 and 16:00
No booking scheduled between 16:00 and 17:00
No booking scheduled between 17:00 and 18:00
No booking scheduled between 18:00 and 19:00
No booking scheduled between 19:00 and 20:00
No booking scheduled between 20:00 and 21:00
No booking scheduled between 21:00 and 22:00
No booking scheduled between 22:00 and 23:00
Enter a selection from the following menu:
1. Enter a room booking
2. Display a booking
3. Display room bookings for the whole day
0. Exit program
3

Event Date (entered as DDMMYYYY): 10102020
---------------
8:00 - 14:00
Event: AGM
Description: Annual meeting of farmers
Contact Information: Oly MacDonald
Phone: 111-222-3333
Farmer's Market
---------------
No booking scheduled between 14:00 and 15:00
No booking scheduled between 15:00 and 16:00
---------------
16:00 - 23:00
Event: Wedding Reception
Description: Reception for Ms. Meadow's daughter and son-in law
Contact Information: Mary Meadows
Phone: 222-333-4444
```

```
---------------

Enter a selection from the following menu:
1. Enter a room booking
2. Display a booking
3. Display room bookings for the whole day
0. Exit program
0

Exiting Room Booking Application
```

Assignment 1 Marking Guide

Requirement	Mark
The submitted zip file is correctly labelled, and contains all project-related files in the package indicated, along with all expected classes, as outlined in the Assignment 1 document in Section IV, Submission Guidelines.	**/4**
Code loads, compiles and executes in Eclipse, with no 'red dots' in the left hand column in Eclipse and no errors generated during execution. Note that failure to submit an executable program may impact your marks in the remainder of this assessment, since most of the marking depends upon being able to test your code by its execution. So even if the underlying code is correct, if it can't be run, it can't be marked—you'll get 0.	**/3**
Implemented all fields, methods and constructors indicated in the UML. (Conversely, you did not implement additional members *not* specified in the UML.) Furthermore, these members have been used appropriately, e.g. you've used the named constants in your switch statement and elsewhere to make your code more readable and robust; each question/response operation is performed by getResponseTo(); findBooking() is used appropriately in other methods, demonstrating good code reuse, etc.	**/8**
Program executes correctly, as demonstrated by having output consistent with the sample output shown in the latest version of the Assignment 1 document. Note that your code must be able to display the output shown *in its entirety*, with no 'Scanner bugs'.	**/8**
Required documentation is provided just inside each class (see Section III(b) of the documentation). Citations for web sites and other sources used in the creation of your code are indicated just before the method(s) in which they appear.	**/3**
Constructors are chained correctly. Private members are never accessed directly outside of the setters and getters.	**/4**
In RoomBooking: toString() chained correctly, i.e. it calls on the toString() methods of the other objects that it is composed of. Output must be identical to that shown, with no '[' or ']' or any other characters added to the output.	**/2**
In TimeBlock: overlaps() correctly determines scheduling conflicts, and this information is reliably used in findBooking()	**/8**
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—simply put, your program is the code equivalent of Donald Trump's presidency—etc.	
Total:	**/40**