CSC 526 — Software Systems Design
Homework #2 Scheduler

Part 1 Due: Monday, 1/30/17, 11:59 PM
Part 2 Due: Wednesday, 2/1/17, 11:59 PM
Part 3 Due: Monday, 2/6/17, 11:59 PM

This assignment focuses on object equality and comparison, cloning, exceptions, and more practice with object-oriented programming. Push your project that includes all source codes, test codes and `myschedule.txt` to GitHub repository called EGR326-HW2 or CSC526-HW2. You can download support files such as `SchedulerGui.java` and `courses.txt` from HERE into your project to run the GUI. Run `SchedulerMain` to launch the program. Note that text files like `courses.txt` should be placed in your IntelliJ project's root folder, the parent of its `src/` or `bin/` subdirectories.

Program Description:

In this program you will write a set of supporting classes for a basic university course scheduling program. The instructor has written the Graphical User Interface that will provide the "front end" or "view" to your program.

The program displays a list of courses loaded from a file `courses.txt` as a table. The table has a column for each weekday (Monday, Tuesday, Wednesday, Thursday, and Friday) and a row for each half-hour interval starting with 8:30 AM. A course is shown in a given cell of the table if it occupies any part of that half-hour interval on that day.

Every course is worth some number of credits from 1 through 5 inclusive. The user's total credits for all courses is shown near the bottom of the window.

The user can click the Add button to add courses to the schedule. It is possible that the user might try to add a course that conflicts with an existing course (a time overlap). In such a case the program, based on the code you will write, displays an error message indicating the conflict and refuses to add the new course.

The user can click a given cell to highlight it, then press the Drop button to remove that course from the schedule. The user can also click the Save button to save a text representation of all courses to the `courses.txt` file on disk. The user can save the schedule to the file in three different orders: by course name, by number of credits, and by start/end time.
To simplify our program, we are going to assume that a given single course takes place at the same time on each day that it occurs. So, for example, no course in our system can meet at both 9:30 AM on Mondays and 10:30 AM on Wednesdays. (If you wanted to emulate such a course in this system, you could add it as two separate courses.)

Your classes are to exactly reproduce the format and overall behavior as specified in this spec. You will have to run the GUI and test individual inputs on your own to verify that your classes are working correctly.

Classes to Implement:

- `Weekday`: an enumeration (`enum`) representing weekdays from Monday-Friday
- `Time`: a class representing times of day such as 12:30 PM or 9:00 AM
- `Course`: a class representing individual courses
- `Schedule`: a class representing a student's current collection of courses
- `ScheduleConflictException`: an exception class to be thrown when course conflicts occur
- `CourseNameComparator`: a class for sorting a schedule by course name
- `CourseCreditComparator`: a class for sorting a schedule by number of credits for each course
- `CourseTimeComparator`: a class for sorting a schedule by days and times

Do <u>not</u> add any *public* methods to these classes other than the ones listed, but you can add your own *private* methods. You may define a `toString` method in any of these classes (you might find that helpful in testing and debugging).

Part 1:

Complete both source and test implementation for Time class/TimeTest and Weekday enum/WeekdayTest

Part 2:

Complete both source and test implementation for Course class/CourseTest

Part 3:

Complete both source and test implementation for all other classes.

Exception Checking:

Your classes should forbid invalid parameters. If any method or constructor is passed a parameter whose value is invalid as per this spec, you should throw an `IllegalArgumentException`. In particular, you must enforce the following:

- No parameter passed to any method (except `equals`) should ever be `null`.
- Time values should have valid numbers of hours (1-12) and minutes (0-59).
- Durations of minutes, such as the duration of a course, must be strictly greater than 0.

For full credit, <u>all methods</u> of all classes should run in a constant amount of time (O(1)) regardless of any parameter value(s) passed, unless otherwise specified. This constraint may affect your choice of implementation and data structures.

(Note that looping over days of the week is still O(1) since there are a fixed number of 5 days in the week.)

Weekday enumeration:

The `Weekday` enumeration represents the five days of the week for courses. It should have the following *public* behavior.

Method	Description
`MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY`	Enumeration values for the five days of the week, declared in this order.
`fromString(`*s*`)`	This *static* method converts a string into the equivalent Weekday enum value. You should match the weekday's full name or its 'short name' (described below), case insensitively. So, for example, `"Monday"`, `"MONDAY"`, `"MoNdAy"`, `"M"`, or `"m"` would all match `MONDAY`. If the string does not match any enum value, throw an `IllegalArgumentException`.
`toShortName()`	Returns a short 1-letter name for the given weekday value, which is the value's first letter (such as `"M"` for `MONDAY`) except for `THURSDAY`, which is `"R"`.
`toString()`	Returns a text representation of this weekday value in title case; rather than the all-uppercase default, produce a string with its first letter capitalized and the remaining letters in lowercase.

Time class:

The `Time` object represents a particular time of day such as 12:30 PM or 9:47 AM. You <u>must</u> ensure that all `Time` objects always have valid state as described in this document. Each `Time` object should have the following *public* behavior.
The `Time` class should also implement the **`Cloneable`** and **`Comparable`**`<Time>` interfaces from `java.lang` so that client code will be able to utilize its `clone` and `compareTo` methods.

Method	Description
`Time(`*hour*`, `*minute*`, `*PM*`)`	Constructor that takes an hour and minute (integers) and a boolean value representing whether it is in the AM (`false`) or PM (`true`) time period.
`fromString(`*str*`)`	This *static* method accepts a string such as `"12:03 PM"` and converts it into a `Time` object, which is returned. Your code must break apart the string in order to extract the individual integers and values such as `12`, `3`, and `"PM"` to interpret their meaning. If the string is in an invalid format, such as missing a colon or AM/PM or number, or if anything else goes wrong, throw an `IllegalArgumentException`.
`clone()`	Returns a copy of the object, following the contract of `clone` from the Java API.
`compareTo(`*time*`)`	Returns an integer indicating this time's placement in the natural ordering of times relative to the given other time, as per the contract of `compareTo` from the Java API Specification. Times are ordered chronologically; for example: 12:00 AM < 8:17 AM < 11:59 AM < 12:00 PM < 3:45 PM < 11:59 PM
`equals(`*o*`)`	Returns `true` if and only if *o* refers to a `Time` object with exactly the same state as this one; otherwise returns `false`.
`getHour(), getMinute()`	Accessors for the time's hour and minute state.

`isPM()`	Returns `true` if this time occurs between 12:00 PM (noon) and 11:59 PM inclusive.
`shift(`*minutes*`)`	Adjusts this time object forward in time by the given number of minutes (an integer). This might cause the object to wrap into the next hour or even wrap around to the next entire day. If the number of minutes is negative, throw an `IllegalArgumentException`. This method can be O(*minutes*) though it doesn't need to be.
`toString()`	Returns a string for this time in HH:MM AM/PM format. The hours and minutes are *always* shown as two-digit numbers, with a colon between and a space between the time and the AM/PM period. For example, `"04:53 AM"` or `"12:07 PM"`.

Course class:

A `Course` object stores information about a particular university course. It has the following *public* behavior.

Method	Description
`Course(`*name*`,` *credits*`,` *days*`,` *startTime*`,` *duration*`)`	Constructor that takes a course name (a string), number of credits (an integer), a Set of weekdays on which the course is offered, the time of day at which the course begins (a time), and its duration in minutes (an integer). In addition to the general argument checks, throw an `IllegalArgumentException` if the set of days is empty or if the credits are not between 1-5 inclusive. You may assume that the set of days is sorted in the natural ordering of weekdays; that no day in the set is `null`; and that no course wraps from one day to the next.
`conflictsWith(`*course*`)`	Returns `true` if this course is in session during any day(s) and time(s) that overlap with the given course. Even a single minute of overlap in the courses is considered to be a conflict. For example, if the two courses both take place on Mondays and one starts at 12:30 PM and last for 60 minutes duration and the other begins at 1:15 PM and lasts for 40 minutes duration, they conflict. This method should run in no worse than O(*duration*) time.
`contains(`*day*`,` *time*`)`	Returns `true` if this course is in session during the given time on the given day. Courses follow the common convention that their start times are inclusive but their ending times are exclusive. In other words, a 60-minute course beginning at 1:30 PM does contain the time of 1:30 PM and it does contain the time of 2:29 PM, but it does not contain 2:30 PM.
`equals(o)`	Returns `true` if and only if *o* refers to a `Course` object with exactly equivalent state as this one; otherwise returns `false`.
`getCredits()`, `getDuration()`, `getName()`, `getStartTime()`	Accessors for the course's various state as passed to the constructor.
`getEndTime()`	Returns the non-inclusive end time for this course, which differs in time by exactly *duration* minutes from the course's start time. For example, if the course begins at 1:30 PM and lasts for 60 minutes, the end time to return is 2:30 PM. This method should run in no worse than O(*duration*) time.
`toString()`	Returns a string representation of this course. The string contains the course's name, credits, days, start time, and duration separated by commas. The days should be shown as a combined string of "short names". For example, if the course occurs on Monday, Wednesday, and Friday, its days should be represented as MWF. An example of the kind of overall string returned would be `"CSE 142,4,MWF,09:30 AM,50"`. Note that this format also matches the expected format of courses in the input file `courses.txt`.

Schedule class:

A `Schedule` object stores information about the collection of courses in which a student is enrolled. It should have the following public behavior. Unless otherwise specified, every method of class `Schedule` should run in O(n) time where n is the number of courses in the schedule. The `Schedule` class should implement the **`Cloneable`** interface so that client code will be able to use its `clone` method. Note that this document does not specify what internal collection or ordering to use for your courses in a schedule. But when *saving* a schedule, you save the courses in the order indicated by the comparator passed to the `save` method.

Method	Description
`Schedule()`	Constructor that creates a new empty schedule. Should run in O(1) time.
`add(`*course*`)`	Adds the given course to this schedule. If the given course conflicts with any of the courses in the schedule as defined previously, a `ScheduleConflictException` should be thrown. *(The GUI's "Add" method calls this.)*
`clone()`	Returns a copy of the object, following the general contract of `clone` from the Java API Specification. In particular, it should be a deep and independent copy, such that any subsequent changes to the state of the clone will not affect the original and vice versa.
`getCourse(`*day*`,` *time*`)`	Returns the course, if any, in this schedule that takes place on the given weekday at the given time. Since courses cannot conflict, there is at most one such course. If no course in this schedule takes place at that time, returns `null`. *(The GUI calls this method for each day at 30-minute intervals to know what to show in the table on the screen. The course must occupy that exact minute in time to be shown.)*
`save(`*printstream*`,` *comparator*`)`	Outputs the courses from this schedule to the given output file (represented as a `PrintStream` object) in the ordering represented by the given course comparator (a class that implements the `Comparator<Course>` interface). The courses should be output one per line in a format consistent with the `toString` behavior of each course. If done properly, the output format will match that of the original `courses.txt` file. See the descriptions of the three course comparator classes later in this document. This method should run in no worse than O(*n* log *n*) time where *n* is the schedule size. *(The GUI's "Save" button calls this with various comparators depending on the user's choice.)*
`totalCredits()`	Returns the total number of credits for which the student is enrolled. For example, if the student is enrolled in a 4-credit class, a 3-credit class, and a 5-credit class, this method would return 12. If the student is not enrolled in any courses, returns 0. This method should run in no worse than O(*n*) time where *n* is the schedule size. *(The GUI calls this in order to display the total credits near the bottom of the window.)*

ScheduleConflictException class:

A `ScheduleConflictException` object is a runtime exception, specific to this application, that indicates that the client has attempted to introduce a conflict into a course schedule (overlapping courses). It has the following public behavior:

Method	Description
`ScheduleConflictException(`*course1*`,` *course2*`)`	Constructor that takes two courses, the ones that conflict, as arguments. Use them to create an appropriate error message. You may assume that *course1* and *course2* are not `null`.

CourseNameComparator, CourseCreditComparator, CourseTimeComparator classes:

An object of any of the three above classes implements the **`Comparator`**`<Course>` interface from `java.util` to help with saving course schedules in different ways. Each class has the following public behavior:

Method	**Description**
`Course____Comparator()`	A zero-argument constructor. (You don't actually need to write a constructor if it would be empty; but you may not add one that requires parameters.)
`compare(`*course1*`,` *course2*`)`	Returns an integer indicating this course's placement in the natural ordering of times relative to the given other time, as per the contract of `compare` from the Java API Specification for `Comparator`. • `CourseNameComparator`: Compares solely by name in alphabetical order. If two courses have the same name, they are considered equal. • `CourseCreditComparator`: Compares first by number of credits in ascending order, breaking ties by comparing names in alphabetical order. If two courses have the same number of credits and same name, they are considered equal. • `CourseTimeComparator`: Compares first by start time in ascending chronological order, breaking ties by ending time (or duration) in ascending chronological order, and finally breaking ties by name in alphabetical order. In other words, if a course begins at an earlier time, it comes earlier in the ordering. If two courses start at the same time, the one that ends sooner comes earlier in the ordering. If two courses have the same start/end time and name, they are considered equal.

Utility.java file (optional):

In a large assignment like this, you may find that you have some common shared code used by many of your other classes. (For example, your error-checking code may be commonly reused.) If so, you can put such code into an optional file `Utility.java` that you will turn in with the others. The contents of this file, if any, are up to you. It should be commented and styled properly like your other files, and you should not place core system behavior in `Utility.java` that is meant to be part of one of the preceding classes. But beyond that it is up to you what, if anything, to use it for.

Development Strategy:

If you try to write all of these classes at once and then compile/run the GUI, you are unlikely to succeed. The GUI is a large system that uses all of your classes in complex ways. A small error in your code will stop it from functioning entirely, giving you poor feedback about what code does and does not work successfully. The GUI also does not call every method of every class nor call them with every possible parameter value.

Therefore it is important to write and test your code incrementally. After adding one method of a class, add one or multiple test cases to test the method you added. Never wait until you implement all methods in class and test. Also, never wait until you implement all classes and then add the test for all classes. For the order of class implementation, we suggest that you write the classes in the order they are listed in this spec, at least for the first few (`Weekday`, then `Time`, then `Course`, then `Schedule`). Remember that you can insert a "stub" version of a particular method or entire class that simply has a pair of empty `{}` braces for the method's body (or simply returns a dummy value like `null` or `-1`). This may allow you to test unfinished classes together.

Unit Testing with JUnit:

You are also <u>required</u> to turn in JUnit test files under tst folder. For each class X, implement XTest class with extensive test cases to test both positive and negative tests for each method including constructor. For example for `Time` class, implement `TimeTest` class which should include not only the basic scenarios but also corner cases. You should have multiple test cases for each method to cover all scenarios. Employing unit tests with JUnit will provide you much simpler, cleaner, and modular way to debug and test than trying to jump right into the GUI, which requires everything to be done before it will run. For each test case, adopt given-when-then (or arrange-act-assert) commenting style whenever possible.

You will be provided with the instructor-written JUnit tests files. You are <u>not</u> allowed to copy the instructor test codes. You should write your own test code which should look different than the provided test code. (If your test code has trivially similar code, then I will consider it as “cheating” per our class policy.) The instructor test code is provided so that you can cross check whether your implementation is indeed correct. **The instructor test code will be released 48 hours prior to each deadline for PART 1 and PART 2 under BB -> HW.**

Hints:

When comparing times, it may help you to internally convert to 24-hour time first to dodge AM vs. PM issues. Consider using `String.format` to help you format times with appropriate leading zeros such as "`04:03 PM`". Recall that some collections can be constructed with comparators as arguments to guide the ordering of elements placed into them. A collection's comparator cannot be changed after it is constructed, but a new collection can be created.

Creative Aspect: myschedule.txt

Along with your program, turn in a file myschedule.txt that represents your <u>actual</u> course schedule this semester. The file's format should match the provided courses.txt, containing one course per line, with name / credits / weekdays /start time / duration separated by commas.

```
EGR 326,3,TR,02:30 PM,90
EGR 424,3,TR,03:45 PM,90
```

For full credit, your file should contain at least 2 courses. If EGR 326 is your only course, make up a second course.

Style and Design Guidelines:

A major focus of this assignment is ensuring that all of your objects are always kept in a valid state. This is accomplished through rigorous argument checking and throwing exceptions on invalid arguments. Part of your grade will be based on whether you handle all of the exceptions and all possible combinations of invalid arguments that could be passed.

Some of your classes will use collections internally to store data. Part of your grade will be based on whether you choose appropriate collections to match the expectations outlined in this spec, such as performance and data ordering. You will also be graded on whether you choose appropriate algorithms to meet the Big-Oh requirements listed in this spec.

Java includes some built-in classes for manipulating dates and times, such as `Date`, `Calendar`, `GregorianCalendar`, and `DateFormat`. You are **forbidden** from using these classes in any way to help you solve this program.

Your code will be graded on whether it follows the **style and design guidelines** taught in lecture. You should also follow a readable and consistent style for your classes such as the one found in the Oracle Java official coding conventions (click here to go to the link).

Redundancy is a major grading focus of every assignment for this course. Some methods are similar in behavior or based off of each other's behavior, so you should call your own methods rather than re-implementing logic. Your classes may have other methods besides those specified, but any other methods you add should be *private*.

Follow **good general style guidelines** such as: making fields `private` and avoiding unnecessary fields (you will lose points if you declare variables as fields that could instead be declared as local variables); declaring collection variables using interface types (e.g. `List` rather than `ArrayList`. Use the parent type, which is more generalized type); initializing fields in constructors, not at declaration; appropriately using control structures like loops and if/else; following "Boolean Zen" (proper use of boolean logic); properly using indentation, good variable names and types; and not having any lines of code wider than 100 characters.

Comment all of your files descriptively in your own words at the top of each class, each method/constructor, and on complex sections of your code. Comments at the top of a class should identify yourself, the assignment / course, and should describe the overall purpose of the class. Method header comments should at a minimum explain the method's behavior, parameters, and return values as appropriate. Also **comment any exceptions thrown** in your method header comments by stating the type of exception and what would cause it to occur. Private helper methods are graded more loosely on commenting than public ones; simply give a private method a brief header explaining its purpose. Also **comment your JUnit test cases using given-when-then (or arrange-act-assert) commenting style**.

For reference, my solution contains roughly 287 "substantive lines" (which excludes things like blank lines, comments, and {} brace lines) according to the class Indenter Tool, though this number is just provided as a sanity check; you do not need to match it or be close to it to get full credit.