CS 1301 – Spring 2021
Project 2, 100 pts.
Due: March 8th at 9:00am

Objectives

- Add data members to classes
- Write methods that use loops and make decisions
- Create and interact with objects
- Test expectations about object state and method return values

Specifications

The project is broken into three parts. Part 1 is worth 74 points. Parts 2 and 3 are worth 14 and 12 points, respectively. Part 2 will be a more difficult than Part 1 and, likewise, Part 3 will be more difficult than Part 2. Part 2 will not be graded if you do not earn 64 of the 74 points on Part 1. Part 3 will not be graded if you do not earn 10 of the 14 points on Part 2.

The specification of each Part is given below. Do not move onto the next part until you have successfully completed ALL of the previous part.

Notes about the project:

- Any project that does not compile will receive a zero.
- For Part 1, you may not discuss the project with anyone within or outside the class. However, you may receive help on Part 1 from the csX lab or the instructor.
- For Parts 2 and 3, the only person that you are allowed to discuss any aspect of these two parts is with the instructor. You may not receive help from any other individual, except the instructor on Parts 2 and 3.

Part 1

Getting Started

1. Download and import `Project2Starter.zip` into Eclipse.

2. Rename the project to be your *FirstnameLastname*Project2, e.g., SallyWhiteProject2. To rename your project, right-click on the project name and select `Refactor` | `Rename`.

   Note: You will lose points if you do not do this. If you do not follow the specified naming convention, five points will be deducted from the score you would have received.

Add a data member to `AlarmClockController`

The following changes must be made in the `AlarmClockController` class.

1. Find and replace `// TODO add a data member` and replace it with a statement that declares an `AlarmClock` instance variable named `theClock`.

2. Find and replace `// TODO: initialize the data member` with a statement that initializes the data member:

    a. instantiate the `this.theClock` data member by calling the `AlarmClock`'s default constructor.

3. Write a getter for the `theClock` data member. Use Eclipse to help you write this method:

    a. Replace the comment `// TODO: write a getter for the data member`. Right-click and select `Source > Generate Getters and Setters` from the shortcut menu.

    b. In the resulting dialog, expand the item next to `theClock`, select `getTheClock()`, and click `Generate`.

    This will give you some code, but we'll need to modify it a little to make it fit with the style that we teach here.

    c. Add an appropriate method specification:

```
/**
 * Gets the AlarmClock object
 *
 * @precondition     none
 * @postcondition    none
 * @return           the AlarmClock object
 */
```

    d. Modify the `return` statement so that it refers to the class data member using **`this.`**:

    e. Save your work and test the constructor

        i. Uncommenting and run the tests in the `TestConstructor` class in the `edu.westga.cs1301.project1.test.alarmclockcontroller` package.

    If you've forgotten how to get Eclipse to help you do this quickly, please refer back to previous exercises.

Write AlarmClockController::addHours()

This method will add a specified number of hours to the controller's clock. It will "roll-over" if the `hours` increases past 23. For example, if the clock currently has an hour of 22 and we call `addHours(5)`, then the `clock`'s `hour` data member will become 3 ***not*** 28. The `AlarmClock` class is already programmed to perform the roll-over; we simply have to call its `incrementHours()` method the appropriate number of times.

Write the following in the `AlarmClockController` class:

1. **Write** the method specification with the following `@`-tags (e.g. `@precondition`, `@precondition`, `@param`):
    a. **description**: Adds the given number of hours to the clock
    b. a precondition where the `hours >= 0`
    c. a postcondition `getClock().getHours() == getClock().getHours()@prev + hours % 24`
    d. a parameter variable `hours` with a description of: `the number of hours to add`
2. **Write** the header for the `addHours()` method. It must satisfy the following requirements:
    a. be accessible to use outside the class (this is your visibility – choose `public` or `private`)
    b. choose the appropriate return type. The method will change something about the object. (the return type can be any valid data type or `void`)
    c. require a single input that represents the number of hours to add. You will only be able to add whole number hours.
3. **Complete** the method's body:
    a. **Enforce the precondition**: write an if-statement that throws a new `IllegalArgumentException` if `hours` violates the condition.
    b. **Write a for-loop** that will execute `hours` times. In the loop body,
        i. call `incrementHour()` on the `theClock` data member
    c. **Uncomment** and **Run** the tests to verify this method in `edu.westga.cs1301.project1.test.alarmclockcontroller.TestAddHours`.

Write AlarmClockController::addMinutes()

Repeat the previous section, this time to write an `addMinutes()` method that adds some number of minutes to the clock. This method must, however, increment the hour if and when the minutes roll-over.

The `AlarmClock` class will ensure the minutes roll-over, but **will not** increment the hour when that happens. Be sure to follow the same development steps outlined for `addHours()` above.

When you have written this method, **uncomment** and **run** the tests in `edu.westga.cs1301.project1.test.alarmclockcontroller.TestAddToMinutes` class. Fix any error(s) in your `AlarmClockController` class that are detected.

Write AlarmClockFormatter::findAmVsPm()

One "best practice" related to the Single Responsibility Principle is that the responsibility for formatting the way an object should look to a user should be separate from that object itself. Thus, an `AlarmClock` is not responsible for how it should look when we print the time to the console. Instead, we are using an `AlarmClockFormatter` class for that responsibility. **Note** that it is in the `views` package, not the `model` package.

The `findAmVsPm()` is method that determines if a clock currently has a morning ("AM") time or an afternoon/evening ("PM") time.

1. **Read** the specifications for this method.
2. **Complete** the body of `findAmVsPm()`
    a. Enforce the precondition
    b. Write code that returns "AM" if the clock's time is currently before 12 hours otherwise it returns "PM".
3. **Run** the tests in `TestFindAmVsPm` and **fix** any errors you find. *These tests are **not** commented out*.

The `findNormalizedHour()` is a method that converts from a 24-hour format to a 12-hour format.

1. **Write** the `findNormalizedHour()` method
    a. **Read** the specification of this method
    b. **Complete** the body of `findAmVsPm()`
        i. Enforce the precondition

c. **Write** code that satisfies the following

   i. Returns `hour-12` if the `hour` is great than 12 (i.e., afternoon)

   ii. Returns 12 if the `hour` is zero (i.e., midnight)

   iii. Returns `hour` otherwise

d. **Run** the appropriate test classes and **fix** any errors you find.

Submitting Part 1

After successfully completing Part 1 of the project. In the package explorer open the **notes.txt** file. If you successfully completed Part 1 of the project then **add** the following line to this file:

```
Successfully completed Part 1 of the project.
```

If you did not complete Part 1, but attempted it state what you did complete and what problems you ran into.

a. Save the `notes.txt` file.

Export the completed Part 1 project to the following filename *FirstnameLastname*Project2Part1.zip and upload to Moodle.

If you do not follow the specified naming convention, five points will be deducted from the score you would have received.

If you successfully completed **all** of Part 1 and you so desire, you may proceed onto Part 2 of the project.

END OF PART 1

Part 2

Add a seconds data member to `AlarmClock`

1. Add a data member to `AlarmClock` that will allow the clock to keep track of current seconds (i.e., 60 seconds per minute)
2. Initialize the seconds to zero in both of `AlarmClock`'s constructors.
3. Write a getter for the new data member
4. Write an `incrementSeconds()` method that increments the seconds by 1, rolling over to 0 if the seconds goes above 59.
5. Uncomment and run the tests in `TestIncrementSeconds` and fix any errors you discover.

Write AlarmClockController::addSeconds()

Write the following in the `AlarmClockController` class:

1. Write an `addSeconds()` method to conform to the following specifications:
    a. has an appropriate method specification
    b. returns nothing
    c. takes one parameter: the number of seconds to add. The number of seconds should be at least zero.
    d. adds that number of seconds the clock. If the seconds rolls over it should increase the minutes by one. If the minutes roll over, it should increase the hours by one.

    Hint: *other `AlarmClockController` method(s) may reduce your workload!*
2. Uncomment and run the appropriate tests and fix any errors you discover.

Write AlarmClockFormatter::formatMinutesInConversationsalTone()

Write the following in the `AlarmClockFormatter` class:

1. Use the specification below and write the method so it implements the specification.

```
/**
 * Formats the time one of the following ways:
 * "It is the top-of-the-hour" (if the minutes is 0)
 * "It is quarter-past the hour" (if the minutes is 15)
 * "It is half-past the hour" (if the minutes is 30)
 * "It is quarter-till the hour" (if the minutes is 45)
 * "It is 23 minutes past the hour" (if the minutes is
 * anything other than 15, 30, or 45; use the correct
 * number of minutes here (23 is just an example)
 *
 * @precondition clock != null
 * @postcondition none
 *
 * @param clock the clock to format
 * @return a string as described above
 */
```

2. Uncomment and run the appropriate tests and fix any errors you discover.

Submitting Part 2

After successfully completing Part 2 of the project. In the package explorer open the `notes.txt` file. If you successfully completed Part 2 of the project then add the following line to this file:

```
Successfully completed Part 2 of the project.
```

If you did not complete Part 2, but attempted it state what you did complete and what problems you ran into.

a. Save the `notes.txt` file.

Export the completed Part 2 project to the following filename *FirstnameLastname*Project2Part2.zip and upload to Moodle.

If you do not follow the specified naming convention, five points will be deducted from the score you would have received.

If you successfully completed all of Part 2 and you so desire, you may proceed onto Part 3 of the project.

END OF PART 2

Part 3

AlarmClockFormatter::formatTimeBasedOnCityTimeZone

Write a method that will return the time in a readable format using the following template:

"The time is {currentTime} in {selectedCity}."

Where {currentTime} is the calculated time and {calculatedCity} is one of the provided cities below. For example, if the time is 12:30 and the city is Atlanta the output string would be:

"The time is 12:30 PM in Atlanta."

Below are the specifications that the method requires:

1. The method will accept an int that represents a city within the continental US time zones
    a. 1 == Atlanta
    b. 2 == Chicago
    c. 3 == Santa Fe
    d. 4 == Sacramento
2. Based on the passed parameter passed when calling the method – the method will calculate the correct time for that city based on the existing alarm clock data.
    a. The method assumes that the stored time data is already based on the EST time zone.
3. Proper pre-conditions should be declared and enforced on the parameter passed into the method:
    a. To determine the correct pre-conditions, look at the acceptable number to city mappings. The pre-condition should enforce that in order to return the time/city info we need valid input.
    b. If the value passed into the method is NOT valid – throw an appropriate exception, IllegalArgumentException.

Add an appropriately named test class. Develop a set of appropriate tests to confirm correct functionality and ensure all tests pass.

AlarmClockFormatter::formatTimeInConversationalTone()

1. Write a method that returns the entire time (hours and minutes) formatted in a friendly human-readable form, using the following rules (assume the current hour is 4):

    - "It is 4 o'clock" when the minutes is 0
    - "It is quarter-past 4" when the minutes is 15
    - "It is half-past 4" when the minutes is 30
    - "It is quarter-till 5" when the minutes is 45
    - "It is 17 minutes past 4" when the minutes (here, 17) is otherwise less than 30
    - "It is 23 minutes till 5" when the minutes (here, 37) is otherwise greater than 30
    - It uses the "normalized hour"

4. Add an appropriately named test class. Develop at least 3 appropriate tests to confirm correct functionality and ensure all tests pass. For example, you could test when the minutes are 0, when the minutes are 30, and when the minutes are at 45.

    In order to get credit, you must choose at least 3 cases to test and then implement and ensure those tests are passing. If there are not at least 3 tests there will be no credit given.

Submitting Part 3

After successfully completing Part 3 of the project. In the package explorer open the `notes.txt` file. If you successfully completed Part 3 of the project then **add** the following line to this file:

```
Also, successfully completed Part 3 of the project.
```

If you did not complete Part 3, but attempted it and Parts 1 and 2 still work correctly, state what you did complete and what problems you ran into.

a. Save the `notes.txt` file.

Export the completed Part 3 project to the following filename *FirstnameLastname*Project2Part3.zip and upload to Moodle.

If you do not follow the specified naming convention, five points will be deducted from the score you would have received.

END OF PART 3

Submission

Submit your completed project to Moodle by the due date.

If you do not follow the specified naming convention, five points will be deducted from the score you would have received.

Grading breakdown

Any project that does not compile will receive a zero.

- 74 pts. – Part 1.
- 14 pts. – Part 2.
- 12 pts. – Part 3.