Com S 227
Fall 2017
Assignment 2
220 points

Due Date: Friday, September 29, 11:59 pm (midnight)

NO LATE SUBMISSIONS

(Remember that Exam 1 is MONDAY, October 2.)

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy acknowledgement* on the Homework page on Blackboard. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

*Note: Our first exam is Monday, October 2, which is just three days after the due date for this assignment. **It will not be possible to have the assignments graded and returned to you prior to the exam**. We can post a sample solution on September 30.*

Please start the assignment as soon as possible and get your questions answered right away!

Introduction

The purpose of this assignment is to give you some practice working with conditional statements. For this assignment you'll create a class called `ClockRadio` that models an alarm clock.

Think about the way a typical alarm clock works. At any given moment, there is a *clock time* (the time of day as normally displayed), an *alarm time*, and also an *effective alarm time* that comes into play when you press the *snooze* button, which is described in more detail below. The clock has buttons that lets you set the clock time and alarm time. It also has a switch with which you can enable the alarm and disable the alarm, and at any given moment it may or may not be *sounding* (making noise). There is a switch that controls whether it will play the radio or the buzzer when it sounds, and there is a way to set whether the display is in 24-hour format or 12-hour format (hours 1-12 with "AM" and "PM").

Note that a `ClockRadio` object is just a model of some aspects of an alarm clock's behavior; it does not actually tell time or buzz out loud! To simulate the passage of time, you'll create a method

```
void advanceTime(int givenMinutes)
```

that simulates moving the current time the given number of minutes forward. To detect whether the alarm is sounding, you'll provide a method

```
boolean isSounding()
```

that returns true if the clock is currently supposed to be sounding.

If the alarm is enabled, then when the clock time reaches or passes the effective alarm time, the clock starts sounding. It remains in the sounding state until the alarm is disabled or the snooze button is pressed.

The snooze feature works as follows: If the snooze button is pressed while the alarm is sounding, it stops sounding and sets the *effective alarm time* to be 9 minutes after the current clock time. In this case we say that "snooze is in effect". If the alarm is not sounding, the snooze button does nothing. Snooze is canceled by either turning off the alarm, resetting the clock time, or resetting the alarm time. The effective alarm time is always equal to the alarm time when snooze is not in effect. Snooze being "canceled" just means that the effective alarm time is reset to be equal to the alarm time again. Pressing the snooze button does not change the alarm time and does not disable the alarm.

Example

Here is a more detailed example. Suppose the current clock time is 5:00 (that is, 5 o'clock in the morning) and the alarm time is 5:15, and now we turn on the alarm. Then suppose we advance the time by 30 minutes. The clock time is now 5:30, and the clock is sounding (in real life, it would have been sounding continuously for 15 minutes). Then we advance the time by 5 minutes more (let the alarm sound for five more minutes), and then press snooze at 5:35. The alarm stops sounding and the *effective* alarm time is now 5:44 (9 minutes in the future). If we advance the time by 5 minutes, the current time is 5:40 and the alarm is not sounding. If we then advance the time by another 4 minutes, to 5:44, the alarm is sounding again. Then suppose we advance the time by 10 minutes (let it sound for 10 minutes) and press snooze again at 5:54, so now the effective alarm time is 6:03. If we advance the time by 30 minutes, to 6:24, the alarm will be sounding. (Most likely, we went and got in the shower without remembering to disable the alarm.) If we disable the alarm, it stops sounding. Doing so also cancels snooze and sets the effective alarm time back to the originally set alarm time of 5:15. (That is, pressing snooze didn't change the alarm time.) If we enable the alarm now, it will sound again the next morning at 5:15, when the time advances another 22 hours and 51 minutes.

Also note: If the alarm is enabled when you set the clock time or set the alarm time, the alarm doesn’t start sounding. For example, if the current time is 3:00 and you set the alarm time for 3:00, it will sound 24 hours later. (Not all real alarm clocks work exactly this way, but this assumption simplifies the implementation.)

Specification of public `ClockRadio` methods

For details on all public methods, see the javadoc online.

In addition to the public methods and two constructors, your class must define the two constants:

```
public static final int SNOOZE_MINUTES = 9;
public static final int MINUTES_PER_DAY = 1440;
```

It's about time...

Before you go any further, make sure you understand how the mod operator ("%") works, since time always “wraps around” when you start the next hour or day. For example, suppose the time is midnight, denoted 00:00 in the 24-hour **`hh:mm`** format, and then 4000 minutes go by. What time is it now? Each 1440 minutes is a full 24 hours, so 4000 % 1440 is 1120, the remaining minutes. This is the number of minutes past midnight. To convert to hours and minutes, 1120 / 60 is 18, and 1120 % 60 is 40, so the time is now 18:40 (aka 6:40 PM). In your implementation, you will want to represent time as the *number of minutes past midnight*, and convert to hours and minutes format only when necessary for the appropriate accessor methods.

Testing and the SpecCheckers

As always, you should try to work incrementally and write tests for your code as you develop it.

Since the test code is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

SpecChecker 1

Your class must conform precisely to this specification. The most basic part of the specification includes the class name and package, the required constants, the public method names and return types, and the types of the parameters. We will provide you with a specchecker to verify that your class satisfies all these aspects of the specification and does not attempt to add any public attributes or methods to those specified. If your class structure conforms to the spec, you should see the message `"2 out of 2 tests pass"` in the console output. (This SpecChecker will not offer to create a zip file for you). Remember that your instance variables should always be declared **`private`**, and if you want to add any additional "helper" methods that are not specified, they must be declared **`private`** as well.

SpecChecker 2

In addition, since this is the second assignment and we have not had a chance to discuss unit testing very much, we will also provide a specchecker that will run some simple functional tests for you. This is similar to the specchecker you used in Assignment 1. It will also offer to create a zip file for you to submit. *Specchecker 2 should be available around the 25th of September. Please do not wait until that time to start testing!*

More about grading

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a third) on the specchecker's functional tests and partly on the grader's assessment of the quality of your code. This means you can get partial credit even if you have errors, and it also means that even if you pass all the specchecker tests you can still lose points. Are you doing things in a simple and direct way that makes sense? Are you defining redundant instance variables? Some specific criteria that are important for this assignment are:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
  - You will lose points for having lots of unnecessary instance variables
  - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.

See the "Style and documentation" section below for additional guidelines.

Style and documentation

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines:

- Each class, method, constructor and instance variable, whether public or private, must have a meaningful and complete Javadoc comment. Class javadoc must include the `@author` tag, and method javadoc must include `@param` and `@return` tags as appropriate.
    - Try to state what each method does in your own words, but there is no rule against copying and pasting the descriptions from this document.
    - Run the javadoc tool and see what your documentation looks like! You do not have to turn in the generated html, but at least it provides some satisfaction :)

- All variable names must be meaningful (i.e., named for the value they store).
- Your code should not be producing console output. You may add `println` statements when debugging, but you need to remove them before submitting the code.

- Do not embed numeric literals in your code. Use the defined constants wherever appropriate.
- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. (A good rule of thumb is: if you had to think for a few minutes to figure out how something works, you should probably include a comment explaining how it works.)
    - Internal comments always *precede* the code they describe and are indented to the same level.
- Use a consistent style for indentation and formatting.
    - Note that you can set up Eclipse with the formatting style you prefer and then use Ctrl-Shift-F to format your code. To play with the formatting preferences, go to Window->Preferences->Java->Code Style->Formatter and click the New button to create your own "profile" for formatting.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder `assignment2`. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag `assignment2`. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "pre" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

Getting started

*Smart developers don't try to write all the code and then try to find dozens of errors all at once; they work **incrementally** and test every new feature as it's written. Here is a rough guide for some incremental steps you could take in writing this class.*

1. Create a new Eclipse project and within it create a package **`hw2`**.

2. Go ahead and create the **`ClockRadio`** class in the **`hw2`** package. Add the two constant declarations and put in stubs for all the methods and constructors described in the Javadoc. For methods that need to return a value, just return a "dummy" value as a placeholder. At this point there should be no compile errors in the project.

3. Import and run the first specchecker. Your code should successfully pass the two tests.

4. Javadoc the classes and methods. This is a required part of the assignment, and doing it now will help clarify for you what each method is supposed to do before you begin the actual implementation.

   Try to write a brief statement in your own words of what the method should do. Copying method descriptions from the online Javadoc is perfectly acceptable, but those are much more detailed than what you are expected to provide. You'll need to fill in the @param and @return tags for completeness.
   Don't forget that the class itself needs a javadoc comment. Your description of the class can be very brief, **but you *must* include an @author tag with your name.**

5. A good thing to start with might be just keeping track of time. That is, think about the methods **`advanceTime(), getClockTimeRaw()`**, and **`setTime()`** (the one-parameter versions). You'll need an instance variable representing the time (as suggested above in the section "It's about time", it is easiest to represent time as the number of minutes after midnight). Start writing some simple statements about what should be observed, and write a usage example for each. For example:

After calling setTime, the clock time should be the value that was set.

```
ClockRadio c = new ClockRadio();
System.out.println(c.getClockTimeRaw()); // expected 0
c.setTime(60);
System.out.println(c.getClockTimeRaw()); // expected 60
```

After calling advanceTime(15), the clock time should be 15 minutes later.

```
c.advanceTime(15);
System.out.println(c.getClockTimeRaw()); // expected 75
```

After calling advanceTime with a number of minutes greater than one day, the clock time should correctly wrap around.

```
c.advanceTime(4000);
System.out.println(c.getClockTimeRaw()); // expected 1195
```

Note that the two-parameter **`advanceTime()`** method does not have to duplicate the code in the one-parameter version! Methods in a class can call each other, so the two-parameter version can just figure out the correct number of minutes and then *call* the one-parameter version. This will be important as you start adding more complex code to the **`advanceTime()`** method.

6. There is a three-parameter version of **`setTime()`** that takes a 12-hour time in the form of hours, minutes, and a boolean indicating AM vs PM. You can work on this method independently at any point, since nothing else depends on it. You don't have to rewrite the logic of the one-parameter **`setTime()`** method - just calculate the correct number of minutes and call it.

7. You can do something similar to step 5 for **`setAlarmTime()`** and **`getAlarmTime()`**. Test your methods. Remember that the constructor needs to initialize the alarm time to 1:00, or 60 minutes past midnight (see the javadoc for the constructors).

8. When you're testing, it might be convenient to print the clock time and alarm time as strings in the form **`hh:mm`** rather than as minutes past midnight. The methods such as

`getClockTimeAsString()` that return a string are also pretty easy. To start out, just use 24-hour format, which is easiest, and worry about the 12-hour format later. Can you use a "helper" method to avoid duplicating the same code three times?

> Here is a very useful trick: once you figure out the value for the hours and minutes, you can create a string in the form **`hh:mm`** like this:
>
> ```
> String timeString = String.format("%02d:%02d", hours, minutes);
> ```
>
> (The funny string with the percent signs is called a "format specifier"; if you are curious how it works, see section 4.3.2 of the text.)

9. You need some way to keep track of whether the alarm is enabled and whether the alarm is currently sounding. Once you do that, you can easily implement **`alarmEnabled()`**, **`alarmDisabled()`**, and **`isSounding()`** (although they won't yet produce correct results, until the next step.) Make sure everything is correctly initialized in your constructor. There are also some related methods **`isBuzzing()`** and **`isPlayingRadio()`** that depend on whether the alarm is currently sounding, *and* on the value set with **`setRadioMode()`**. You can do these independently of everything else.

10. Now the fun part. Modify your **`advanceTime()`** method so that when the clock time reaches or passes the alarm time, the clock starts sounding. (Later, to implement snooze, you'll need to distinguish between the set alarm time and the effective alarm time, but you can just assume they're the same for now. Don't worry about snooze yet!) As always, start with some test cases. For example, what should the output be for the following?

If the alarm is enabled, advancing the time past the alarm time should cause the alarm to sound.

```
c = new ClockRadio();
c.setTime(300);        // 5:00
c.setAlarmTime(315);   // 5:15
c.alarmEnabled();
System.out.println(c.isSounding()); // expected false
c.advanceTime(30);
System.out.println(c.isSounding()); // expected true
```

If the alarm is sounding, disabling the alarm should cause it to stop sounding.

```
c.alarmDisabled();
System.out.println(c.isSounding()); // expected false
```

Does your logic still work when the alarm time is a smaller number than the current clock time (i.e., you advance the time past midnight)? What about the other way around? Here's an example to try.

```
c.setTime(1430);    // 23:50, or 11:50 PM
c.setAlarmTime(5);  // 00:05, or 12:05 AM
c.alarmEnabled();
System.out.println(c.isSounding()); // expected false
c.advanceTime(5);
System.out.println(c.isSounding()); // now 23:55, expected false
c.advanceTime(5);
System.out.println(c.isSounding()); // now 00:00, expected false
c.advanceTime(5);
System.out.println(c.isSounding()); // now 00:05, expected true
```

11. At this point everything should be working except the snooze feature, which is definitely the hardest part. Be sure that you have unit tests for everything you've done so far, because implementing snooze will modify several existing methods and you want to check that you haven't broken the things that were working before! Be sure to keep a backup copy of what you've done so far too. *If everything is correct except the snooze feature, you will lose at most 30 points.*

First, you'll need to keep track of the *effective alarm time* (which is affected by snooze) as opposed to the alarm time (which is only set by the `setAlarmTime()` method). Initially, they're the same:

Setting the alarm time should set the effective alarm time to the same value.

```
c = new ClockRadio();
c.setTime(300);       // 5:00
c.setAlarmTime(315);  // 5:15
System.out.println(c.getAlarmTimeRaw());  // expected 315
System.out.println(c.getEffectiveAlarmTimeRaw());  // expected 315
```

Carefully modify your `advanceTime()` method so that it uses the effective alarm time rather than the alarm time to determine whether the alarm should sound. Re-run all your tests and make sure everything still works.

To implement the snooze feature, read the javadoc carefully and think about what behavior you should observe. Reread the example scenario on page 2. Here are some examples of simple behaviors you might observe:

If snooze() is called when the alarm is sounding, the alarm stops sounding.

```
c = new ClockRadio();
c.setTime(300);       // 5:00
c.setAlarmTime(315);  // 5:15
c.alarmEnabled();
System.out.println(c.isSounding()); // expected false
c.advanceTime(30);
System.out.println(c.isSounding()); // expected true
c.advanceTime(5);
c.snooze();
System.out.println(c.isSounding()); // expected false
```

Calling snooze() alters the effective alarm time, but not the alarm time.

```
System.out.println(c.getClockTimeRaw()); // expected 335
System.out.println(c.getAlarmTimeRaw()); // expected 315
System.out.println(c.getEffectiveAlarmTimeRaw());  // expected 344
```

After snooze(), advancing the time past the effective alarm time causes the alarm to sound.

```
c.advanceTime(5);
System.out.println(c.isSounding()); // expected false
c.advanceTime(4);
System.out.println(c.isSounding()); // expected true
```

After disabling the alarm, the effective alarm time should be the same as the alarm time.

```
c.alarmDisabled();
System.out.println(c.getClockTimeRaw()); // expected 344
System.out.println(c.getAlarmTimeRaw()); // expected 315
System.out.println(c.getEffectiveAlarmTimeRaw());  // expected 315
```

A key point to notice is that the alarm time and the effective alarm time are different ONLY when snooze is in effect.

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Homework page on Blackboard, before the submission link will be visible to you.

Please submit, on Blackboard, the zip file that is created by the second SpecChecker. The file will be named **`SUBMIT_THIS_hw2.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`hw2`**, which in turn contains one file, **`ClockRadio.java`**.

Please LOOK at the zip file you upload and make sure it is the right one!

Submit the zip file to Blackboard using the Assignment 2 submission link and **verify that your submission was successful by checking your submission history page**. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under “Syllabus, office hours, useful links.”

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw2**, which in turn should contain the file **`ClockRadio.java`**. Make sure all files have the extension **`.java`**, NOT **`.class`**. You can accomplish this easily by zipping up just the **src** directory of your project (NOT the entire project). The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and **not** a third-party installation of WinRAR, 7-zip, or Winzip.