Com S 227
Fall 2022
Assignment 1
120 points

Due Date: Friday, September 16, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Sept 15)

10% penalty for submitting 1 day late (by 11:59 pm Sept 17)

No submissions accepted after Sept 17, 11:59 pm

This assignment is to be done on your own. See the Academic Integrity policy in the syllabus, for details.
You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. 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.

Please start the assignment as soon as possible and get your questions answered right away. It is physically impossible for the staff to provide individual help to everyone the afternoon that the assignment is due!

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. See the "More about grading" section.

Contents

Tips from the experts: How to waste a lot of time on this assignment

1. Start the assignment the night it's due. That way, if you have questions, the TAs will be too busy to help you and you can spend the time tearing your hair out over some trivial detail.
2. Don't bother reading the rest of this document, or even the specification, especially not the "Getting started" section. Documentation is for losers. Try to write lots of code before you figure out what it's supposed to do.
3. Don't test your code. It's such fun to remain in suspense until it's graded!

Overview

The purpose of this assignment is to give you some practice with the process of implementing a class from a specification and testing whether your implementation conforms to the specification. You'll also get some practice with modular arithmetic.

For this assignment you will implement one class, called **`CarStereo`**, that models the behavior of a simple radio. A radio is constructed with a given range of frequencies to which it can be tuned, called its *band*. Within that range, there is a possible number N of *stations* it can be tuned to, represented simply as integers 0 through N - 1. The radio also has a *volume*, which is a number between 0.0 and 1.0, adjusted in increments of 0.16 (defined in the code by the constant **`VOLUME_STEP`**).

A given "band" of frequencies, such as the AM or FM bands you are familiar with, is a range of possible broadcast frequencies within a specified minimum and maximum. Radio stations can broadcast on a frequency in that range, but the frequencies have to be spread out so that they don't overlap. As is common in the real world, we assume that:

- the band is divided up into N intervals of equal width, and
- the broadcast frequency for each possible station is the **midpoint** of the corresponding interval.

However, the frequency to which the radio is tuned may not exactly match the broadcast frequency of any station. In that case, we say that the radio's *current station* is the one whose broadcast frequency is closest to the radio tuner frequency.

Here is a simple example. Suppose the minimum is 100 and the maximum is 200 and there are N = 5 stations. Then stations numbered 0 through 4 broadcast on frequencies 110, 130, 150, 170, and 190, respectively. So to set the tuner to "station 3", you'd use a calculation like the one below to determine that its exact broadcast frequency is 170:

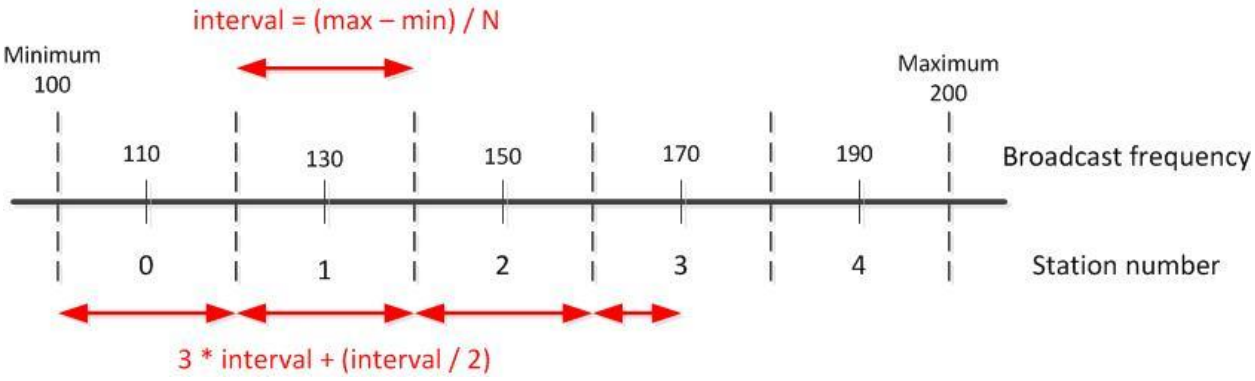

Figure 1 - converting station number to frequency

Conversely, if you turn the dial and set the tuner to frequency 162, you'll be listening to station 3 at frequency 170, since that is closest to 162. The illustration below shows one idea for how to calculate this. The convention is that if the tuner is exactly halfway between stations (e.g. at 160.0), then it the current station is the next *highest* one (in that case station 3).

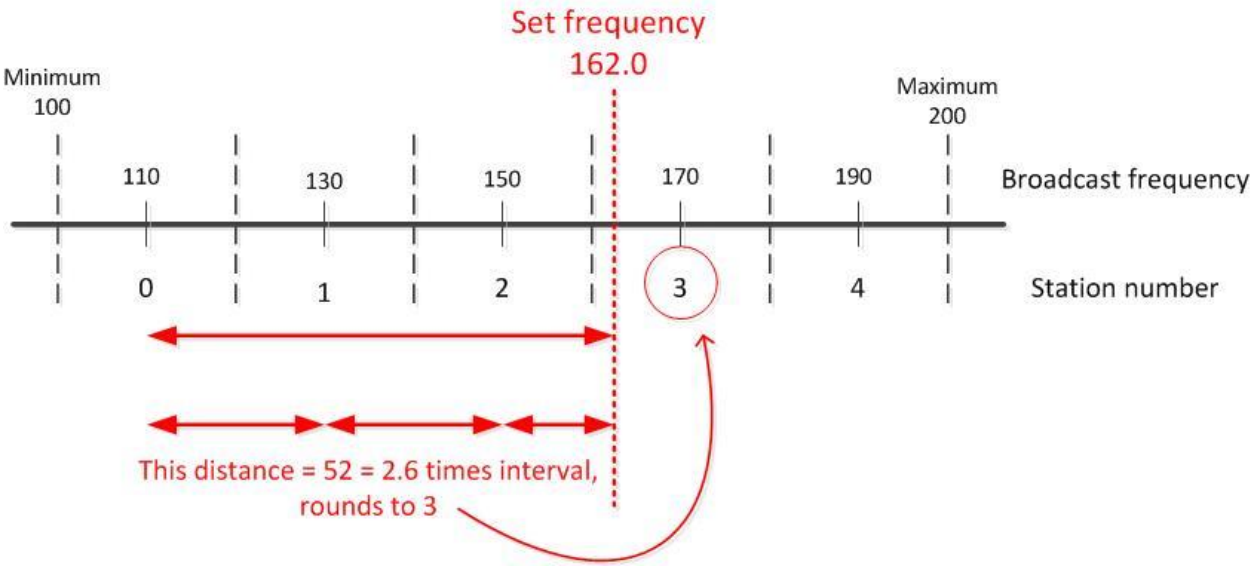

Figure 2 - converting frequency to station number

There are several ways the radio can be tuned:

- Directly setting the tuner frequency to an exact value, represented by the method **`setTuner(double)`**. (This is possibly not very realistic but is useful for testing.)
- Turning a dial, represented by the method **`turnDial(double)`**. We assume that one full rotation takes the tuner from the minimum frequency to the maximum frequency and that fractions of a rotation behave proportionally; however, in no case is the tuner set above the maximum nor below the minimum.
- Using a *seek* function, represented by the methods **`seekUp`** and **`seekDown`**. If the current tuner value corresponds to station *k*, then **`seekUp()`** sets the tuner to the broadcast frequency of station number *k + 1* (wrapping around to zero if *k* is N – 1); **`seekDown()`** sets the tuner to the broadcast frequency of station number *k - 1* (wrapping around to N - 1 if *k* is 0).
- Pressing a *preset* button, represented by the method **`goToPreset`**. The preset is one of the station numbers 0 through N – 1, and this method sets the tuner to be that station's exact broadcast frequency.

Please note that you should not use any conditional statements (i.e. "if" statements), or anything else we haven't covered, for this assignment. There will be a couple of places where you need to choose the larger or smaller of two numbers, which can be done with the methods **`Math.max()`** or **`Math.min()`**. For the wrapping behavior of the station numbers, you can use the mod (`%`) operator. (*You may be penalized slightly for using conditional statements, because you are just being lazy and making your code longer and more complicated. As a point of reference, our prototype has 42 lines of code other than method headings, curly braces, and comments.*)

Specification

The specification for this assignment includes this pdf along with any "official" clarifications announced on Canvas.

There is one public constant:

```
public static final double VOLUME_STEP = 0.16;
```

There is one public constructor:

```
public CarStereo(double givenMinFrequency, double givenMaxFrequency,
    int givenNumStations)
```

Constructs a new radio with the given band and number of stations. Initially the volume is 0.5, the tuner is at the minimum frequency, and the preset is station 0.

There are the following public methods:

`public double getVolume()`

Returns the current volume.

`public void louder()`

Increases the volume by VOLUME_STEP, but not going above 1.0.

`public void quieter()`

Decreases the volume by VOLUME_STEP, but not going below 0.

`public double getTuner()`

Returns the frequency to which this radio is currently tuned.

`public void setTuner(double givenFrequency)`

Sets the tuner frequency to the given value if possible. If the given value is above the maximum, the tuner is set to the maximum, and if it is below the minimum, the tuner is set to the minimum.

`public void turnDial(double degrees)`

Adjusts the current tuner frequency by turning a knob by a given amount, specified as degrees of rotation (positive or negative). A full rotation corresponds to the full band width (maximum frequency - minimum frequency), and any fraction of a rotation adjusts the frequency by the same fraction of the band width.

`public void setTunerFromStationNumber(int stationNumber)`

Sets the tuner to be the broadcast frequency of the given station if possible. If the given station number is below zero, the tuner is set to the broadcast frequency of station 0, and if the given station number is greater than or equal to N, sets the tuner to the broadcast frequency of station N – 1, where N denotes the number of stations.

`public int findStationNumber()`

Determines the station number that is closest to the radio's current tuner frequency. If the tuner is at exactly the midpoint between two stations, rounds to the higher station number; however, if the tuner is at the maximum frequency (which would normally round to station N) then N – 1 is returned (where N denotes the number of stations).

`public void seekDown()`

Sets the tuner to the broadcast frequency of the next station below the current one, wrapping around to station N – 1 if the current station is 0 (where N denotes the number of stations).

`public void seekUp()`

Sets the tuner to the broadcast frequency of the next station above the current one, wrapping around to zero if the current station number is N – 1 (where N denotes the number of stations).

`public void savePreset()`

Stores the current station number as the preset.

`public void goToPreset()`

Sets the frequency to be the broadcast frequency of the current preset station.

Where's the main() method??

There isn't one! Like most Java classes, this isn't a complete program and you can't "run" it by itself. It's just a single class, that is, the definition for a type of object that might be part of a larger system. To try out your class, you can write a test class with a main method like the examples below in the "getting started" section.

There is also a specchecker (see below) that will perform a lot of functional tests, but when you are developing and debugging your code at first, you'll always want to have some simple test cases of your own, as in the getting started section below.

Suggestions for 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. Since this is our first assignment, here is an example of some incremental steps you could take in writing this class.*

0. Be sure you have done and understood Lab 2.

1. Create a new, empty project and then add a package called **`hw1`**. ***Be sure to choose "Don't Create" at the dialog that asks whether you want to create module-info.java***.

2. Create the **`CarStereo`** class in the **`hw1`** package and put in stubs for all the required methods, the constructor, and the required constant. Remember that everything listed is declared **`public`**. For methods that are required to return a value, just put in a "dummy" return statement that returns zero or false. There should be no compile errors. ***DO NOT COPY AND PASTE directly from this pdf!*** *This leads to insidious bugs caused by invisible characters that are sometimes generated by sophisticated document formats.)*

3. Briefly javadoc the class, constructor and methods. This is a required part of the assignment anyway, and doing it now will help clarify for you what each method is supposed to do before you begin the actual implementation. (Repeating phrases from the method descriptions here or in the Javadoc is perfectly acceptable; however, ***DO NOT COPY AND PASTE directly from this pdf!*** *This leads to insidious bugs caused by invisible characters that are sometimes generated by sophisticated document formats.)*

4. Look at each method. Mentally classify it as either an *accessor* (returns some information without modifying the object) or a *mutator* (modifies the object, usually returning **`void`**). The accessors will give you a lot of hints about what instance variables you need.

5. You might start with the volume-related methods, since they are pretty simple. Before writing any code, however, create a separate class with a main method, and come up with a simple usage example or test case, something like this:

```
import hw1.CarStereo;
public class SimpleTests
{
  public static void main(String[] args)
  {
    CarStereo c = new CarStereo(100, 200, 5);
    System.out.println(c.getVolume());  // expected 0.5
    c.louder();
    c.louder();
    System.out.println(c.getVolume());  // expected 0.82 (*)
    c.louder();
    c.louder();
    System.out.println(c.getVolume());  // expected 1.0
    c.quieter();
    System.out.println(c.getVolume());  // expected 0.84
  }
}
```

(*) Note: you may see something confusing like **`0.8200000000000001`** . Do not worry. We have not talked about this in detail in class, but this is normal, because floating-point arithmetic is never exact. If you are within a few decimal places of the expected value your code is probably fine.

The presence of the accessor method **`getVolume`** suggests that you likely need an instance variable to store the current volume. Remember to initialize it correctly in the constructor!

6. Next you could think about the tuner frequency. Again, the presence of the method **`getTuner`** suggests that we probably need an instance variable for the current frequency. In order to test it we need some way to set the tuner. As seen in the introduction, there are several ways to do this, but the simplest seems to be the method **`setTuner`**, so let's implement that. Try writing a usage example:

```
c = new CarStereo(100, 200, 5);
System.out.println(c.getTuner());  // expected 100.0
c.setTuner(123.4);
System.out.println(c.getTuner());  // expected 123.4
c.setTuner(500);
System.out.println(c.getTuner());  // expected 200.0
c.setTuner(42);
System.out.println(c.getTuner());  // expected 100.0
```

Notice that if you try to set a frequency that is outside the range [minimum, maximum] given in the constructor, the resulting value is clamped to that range. (Remember, you don't need a conditional for this! Use Math.max and Math.min.). But in order to do this, you need access to the minimum and maximum frequency, suggesting that those two constructor parameters need to be stored as instance variables so that those values are available to other methods.

7. The next option for setting the tuner would be the method **`turnDial`**. Again start with a usage example to make sure the desired behavior is absolutely clear, before you waste time writing any code:

```
c = new CarStereo(100, 200, 5);
System.out.println(c.getTuner());  // expected 100.0
c.turnDial(360);                   // full rotation
System.out.println(c.getTuner());  // expected 200.0
c.turnDial(-180);                  // half a rotation to left
System.out.println(c.getTuner());  // expected 150.0
c.turnDial(15);                    // 1/24th rotation right
System.out.println(c.getTuner());  // expected ~154.17
c.turnDial(720);                   // two rotations right
System.out.println(c.getTuner());  // expected 200.0
```

8. The remaining methods all require us to convert between the *frequency* and the *station number*. You might be tempted to look at an accessor method such as **`findStationNumber`** and create an instance variable for the station number. That is a Bad Idea ™, though. Why? You already have an instance variable for the tuner frequency, and you can determine the station number from the frequency. An instance variable for the station number would be redundant, and *redundant instance variables almost always lead to errors*.

So instead, start with the slightly simpler method **`setTunerFromStationNumber`**. First, carefully study Figure 1 and do the calculation by hand. Then write a usage example:

```
c = new CarStereo(100, 200, 5);
c.setTunerFromStationNumber(3);
System.out.println(c.getTuner());  // expected 170.0
c.setTunerFromStationNumber(42);
```

```
System.out.println(c.getTuner());  // expected 190.0
c.setTunerFromStationNumber(-5);
System.out.println(c.getTuner());  // expected 110.0
```

9. Next study Figure 2 and do the calculation by hand to make sure it's clear to you what to do in general. Write a usage example:

```
c = new CarStereo(100, 200, 5);
c.setTuner(162);
System.out.println(c.findStationNumber()); // expected 3
c.setTuner(134);
System.out.println(c.findStationNumber()); // expected 1
c.setTuner(180);
System.out.println(c.findStationNumber()); // expected 4
c.setTuner(200);
System.out.println(c.findStationNumber()); // expected 4
```

(The latter example, where the tuner is set to the max value, might seem like a kind of special case requiring a conditional. But remember, it is always correct to use Math.min to clamp the station number down to N – 1, where N is the number of stations.)

10. Once those methods are working correctly, you can start on the rest. For example, try **`seekUp`**. Start with a simple example:

```
c = new CarStereo(100, 200, 5);
c.setTuner(162);
c.seekUp();
System.out.println(c.getTuner());  // expected 190
```

Why 190? The spec says "*Sets the tuner to the broadcast frequency of the next station above the current one...*" If the tuner is at 162, the current station number is 3 (according to **`findStationNumber`**), so **`seekUp`** should set the tuner to the exact broadcast frequency of station 4, which is 190 (as determined by **`setTunerFromStationNumber`**). Try another one:

```
c.setTuner(185);
c.seekUp();
System.out.println(c.getTuner());  // expected 110
```

Why 110? The spec continues, "…*wrapping around to 0 when the current station is station N – 1 (where N is the number of stations).*" If the tuner is at 185, the current station number is 4, and 4 + 1 is 5, so we wrap around to zero, and the broadcast frequency for station 0 is 110 for this radio. For the wrapping behavior, you can just mod by the number of stations, after adding 1, e.g.,

new station number = (old station number + 1) % number of stations

11. The method **`seekDown`** is similar, with one little twist. The issue is the way that Java handles the use of the mod operator with a negative number. Using N = 5 as an example, when the station number is zero, we want to be able to subtract 1 and wrap around to 4. But the expression **`-1 % 5`** has value -1, not 4. An easy way to fix this is just to add N when subtracting 1, which forces the remainder to be positive, e.g., **`(0 + 5 – 1) % 5`** does have value 4. In general:

new station number = (old station number + number of stations - 1) % number of stations

And of course, try a test case or two:

```
c.setTuner(98);
c.seekDown();
System.out.println(c.getTuner());  // expected 190
```

12. Finally implement **`savePreset`** and **`goToPreset`**, starting with a simple test case:

```
c = new CarStereo(100, 200, 5);
c.setTuner(162);
c.savePreset();    // save station 3
c.setTuner(123.4); // whatever
c.goToPreset();    // tune to station 3's broadcast frequency
System.out.println(c.getTuner());  // expected 170
```

The important thing to notice is that **`goToPreset`** is specified to jump to the exact broadcast frequency for the saved station number, which may be different than the tuner frequency when **`savePreset`** was called.

The SpecChecker

You can find the SpecChecker on Canvas. Import and run the SpecChecker just as you practiced in Lab 1. It will run a number of functional tests and then bring up a dialog offering to create a zip file to submit. Remember that error messages will appear in the *console* output. There are many test cases so there may be an overwhelming number of error messages. ***Always start reading the errors at the top and make incremental corrections in the code to fix them.*** When you are happy with your results, click "Yes" at the dialog to create the zip file. See the document “SpecChecker HOWTO”, link #12 on our Canvas front page, if you are not sure what to do.

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 TA'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? Are you using a conditional statement when you could just be using **`Math.min`** or modular arithmetic? Are you using a loop for something that can be done with integer division? 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 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, in addition to the two specific bullet points above.

- **Each class, method, constructor and instance variable, whether public or private, must have a meaningful javadoc comment**. The javadoc for the class itself can be very brief, but must include the **`@author`** tag. The javadoc for methods must include **`@param`** and **`@return`** tags as appropriate.
    - Try to briefly state what each method does in your own words. However, there is no rule against repeating the descriptions from the documentation. *However: **do not literally copy and paste from this pdf**! This leads to all kinds of weird bugs due to the potential for sophisticated document formats like Word and pdf to contain invisible characters.*
    - 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.

- 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 an internal comment explaining how it works.)

    - Internal comments always *precede* the code they describe and are indented to the same level. In a simple homework like this one, as long as your code is straightforward and you use meaningful variable names, your code will probably not need any internal comments.

- 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 **`hw1`**. 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 **`hw1`**. *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. (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 announcements from the instructors on Canvas 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 Canvas. (We promise that no official clarifications will be posted within 24 hours of the due date.)

What to turn in

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

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw1.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw1`**, which in turn contains one file, **`CarStereo.java`**. Please **LOOK** at the file you upload and make sure it is the right one!

Submit the zip file to Canvas using the Assignment 1 submission link and **VERIFY** that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", link #11 on our Canvas front page.

> 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 **hw1**, which in turn should contain the file **CarStereo.java**. You can accomplish this by zipping up the **src** directory of your project. **Do not zip up 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. The Assignment Submission HOWTO includes detailed instructions for creating a zip file if you need them.