Com S 227
Spring 2021
Assignment 2
200 points

Due Date: Tuesday, March 2, 11:59 pm (midnight)

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

10% penalty for submitting 1 day late (by 11:59 pm March 3)

No submissions accepted after March 3, 11:59 pm

(Remember that Exam 1 is THURSDAY, March 4.)

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 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.

Note: Our first exam is Thursday, March 4, which is just two 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****.*

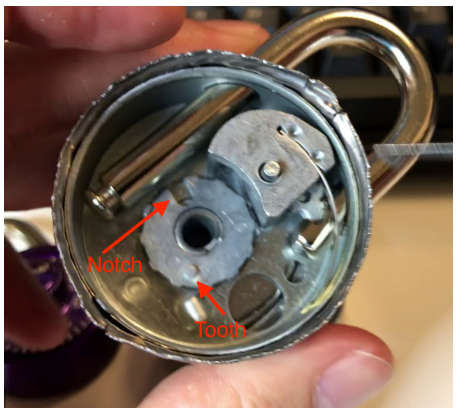
Please start the assignment as soon as possible and get your questions answered right away!

Introduction

The purpose of this assignment is to give you lots of practice working with conditional logic. And yes, modular arithmetic! For this assignment you'll create one class called `Padlock` that is a model of a typical combination lock with a rotating dial, found in schools and locker rooms everywhere. This is a challenging (and fun) problem that will probably seem confusing at first unless you

start adopting a *test-driven* and *incremental* development strategy. The "Getting Started" section offers some suggestions on how to go about this. Don't try to solve it in your head and write all the code at once!

Overview



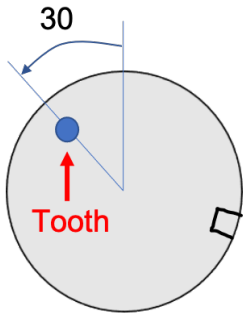
The lock mechanism consists of three discs mounted on the same axis. Each disc has a notch on its outer edge, and when the discs are rotated so that the three notches are aligned in the right spot, the lock can be opened. The front disc, aka disc 3, is connected directly to a dial, which we assume is labeled around its outer edge from 0 to 359, ordered clockwise, corresponding to degrees of rotation.

The front disk has a small protrusion, or "tooth", on its back side, and the middle disc (disc 2) has a tooth on both its front and back, and the rear disk (disc 1) has a tooth on its front side. The illustration at right shows the inside of the lock with the dial, disc 3, and disc 2 removed; you can see the notch and the tooth on disc 1.

Only the front disc (disc 3), connected to the dial, can be turned directly by the user, but if turned far enough the tooth will bump into the tooth on disc 2 (the middle one) and force it to turn as well. Likewise, if turned far enough the tooth on the back of disc 2 will contact the tooth on the disc 1 (the rear disc) and cause it to turn. Thus, after turning the dial two complete revolutions clockwise, it will be possible to push disc 1 into any desired position. After that, if the dial is turned counterclockwise, discs 1 and 2 will initially remain stationary. After one complete revolution counterclockwise, the tooth on the front disc will again contact the tooth on disc 2 and cause it to rotate counterclockwise, still leaving the rear disc alone. In this way disc 2 can be rotated into a desired position. Finally, the front disc can be rotated clockwise to a desired position without affecting the other two.

Breaking into people's lockers is evidently quite a thing these days, because there is a ton of information available on the internet about how these locks work and how to defeat them. Here is a great computer animation: https://www.youtube.com/watch?v=sftkP4CjjZs .
(However, note that in our model we are assuming that both the front and back teeth for disc 2 are in the same position, unlike the animation). And here is a video of a similar lock constructed using transparent materials. (This one actually has four discs, but the idea is the same.)
https://interestingengineering.com/this-transparent-combination-lock-model-clearly-explains-what-goes-inside

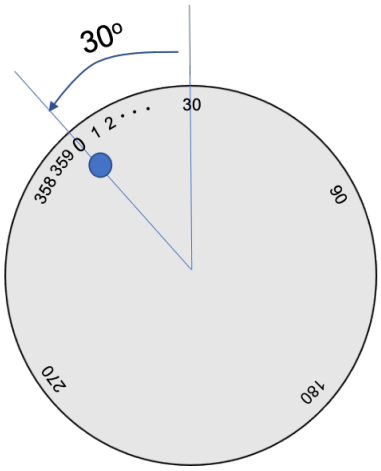
We describe the *position* of each disc as the number of degrees that its tooth is rotated counterclockwise from the zero position at the top. In general, we require the position angle to be "normalized" so it is always a number in the range from 0 through 359. For the front disc, its position always matches the number at the top of the dial; i.e., we assume the tooth is right behind the number 0. Each disc also has an *offset*, in degrees, which is based on the angle between its tooth and the notch. If the notch is in the correct final alignment position when the disc's position is D degrees, then we say the offset is D. Notice we don't care what the actual angle is between the tooth and the notch; we care what the offset is: when every disc's current position is equal to its offset, that means the notches are all correctly aligned and the lock can be opened.



A disc at position 30 degrees

The values of these offsets are determined by the combination. The way we interpret a combination X, Y, Z is according to the package directions, like this:

- rotate the dial two complete revolutions clockwise and then stop at X
- rotate the dial one complete revolution counterclockwise and then stop at Y
- rotate the dial clockwise and stop at Z



When the front disc is at position 30 degrees, the number 30 is at the top of the dial.

It would be convenient if when the combination is X, Y, Z the offsets for discs 1, 2, and 3, respectively, would simply be X, Y, and Z. However, if we turn the dial several complete revolutions counterclockwise and stop when the dial is at, say, 100 degrees, the other two discs will not be rotated 100 degrees. We also have to account for the width of the tooth itself. If, for example, the tooth is wide enough to occupy two degrees worth of the disc's rotation, then in this scenario our disc 2 would be at 102 degrees and disc 1 would be at 104 degrees. Similarly, if we then rotate the dial clockwise a full revolution and stop at 100 degrees, disc 2 would be at 98. In general to get combination X, Y, Z, the offsets would be X - 2 * TOOTH for disc 1, Y + TOOTH for disc 2, and Z for disc 3, where TOOTH is the width of the tooth, expressed in degrees.

API for the Padlock class

Here is a brief summary of the required methods. *Further details are found in the online Javadoc, which is considered to be part of this specification!*

There is one public constant, declared as follows, representing the width of the tooth on each disc, expressed in degrees of rotation.

```
public static final int TOOTH = 2;
```

There is one constructor:

`Padlock(int n1, int n2, int n3)`
Constructs a padlock whose combination will be n1, n2, n3

There are the following public methods:

`void close()`
closes the lock, regardless of whether the discs are aligned

`int getDiscPosition(int which)`
returns the current position of the given disc (1, 2, or 3)

`boolean isAligned()`
returns true if the three discs are aligned in position for the lock to open

`boolean isOpen()`
returns true if the lock is currently open

`void open()`
opens the lock, if possible

`void randomizePositions(Random rand)`
sets the disc positions to randomly generated, valid values

`void setPositions(int n1, int n2, int n3)`
sets the disc positions to the given values (as closely as possible)

`void turn(int degrees)`
turns the dial the given number of degrees, where a positive number indicates a counterclockwise rotation and a negative number indicates a clockwise rotation

`void turnLeftTo(int number)`
turns the dial counterclockwise until the given number is at the top

`void turnRightTo(int number)`
turns the dial clockwise until the given number is at the top

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.**

Since this is the second assignment and we have not had a chance to discuss unit testing very much, we will 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. The *specchecker should be available around the 25th of February. Please do not wait until that time to start testing! You can use the examples in the Getting Started section as a starting point.*

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.
      However: ***do not literally copy and paste from this pdf*** *or the online Javadoc! This leads to all kinds of weird bugs due to the potential for sophisticated document formats like Word and pdf to contain invisible characters.*

- 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.
- Try not to embed numeric literals in your code. In particular, use the TOOTH constant where appropriate, not the literal 2. Your code should still work correctly if the TOOTH value is changed!
- 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 **`hw2`**. 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 **`hw2`**. *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 "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

If you have a question that absolutely cannot be asked without showing part of your source code, change the visibility of the post to "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

*Remember to work **incrementally** and test new features as you implement them. Since this is only our second assignment, 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`**. Create the **`Padlock`** 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. **There should be no compile errors**. You should be able to run the sample main class SimpleTest.java.

2. 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. (Copying from the method descriptions here or in the Javadoc is perfectly acceptable; however, ***DO NOT COPY AND PASTE DIRECTLY from the pdf or online Javadoc!*** *This leads to insidious bugs caused by invisible characters that are sometimes generated by sophisticated document formats. If you are super lazy about typing, it should be ok if you copy, paste into a dumb text editor like Notepad, and then copy/paste from there.)*

3. You could start by first making sure you can set and get the current rotation of each disc. According to the constructor, assuming that the tooth width is 2 degrees, the initial position of the discs is always 0 degrees for disc 3, 2 degrees for disc 2, and 4 degrees for disc 1. (This is true regardless of the constructor arguments.). So you should be able to set up a simple test class with the following:

```
Padlock p = new Padlock(10, 20, 30);
printPositions(p);                  // expected 4 2 0
p.setPositions(42, 137, 17);        // order is disc 1, 2, 3
printPositions(p);                  // expected 42 137 17
```

where **`printPositions`** is a helper method that just uses the **`getDiscPosition`** method to display the three angles:

```
private static void printPositions(Padlock p)
{
  int c = p.getDiscPosition(3);
  int b = p.getDiscPosition(2);
  int a = p.getDiscPosition(1);
  System.out.println(a + " " + b + " " + c);  // back to front (1, 2, 3)
}
```

This would also be a good time to make sure you can normalize the angles so they are always between 0 and 359, for example:

```
  p.setPositions(-90, 800, 42);
  printPositions(p);                    // expected 270 80 42
```

(Tip: the sample code shown above can be found in the given file SimpleTest.java)

3. At this point you could either continue working on disc rotations, or implement the lock logic; the two tasks can be tackled independently. Suppose we do lock logic first. The key piece is determining whether the discs are "aligned", as specified in the **`isAligned`** method. For that we need to know the offset for each disc, that is, the angle of rotation that will put its notch in the right position for the lock to open. According to the general description of the lock internals, if the combination is 10, 20, 30, then the offsets are 6, 22, and 30, for discs 1, 2, and 3, respectively. The **`isAligned`** method should return true if the current positions of all three discs match their offsets.

So, the behavior we expect is something like this (according to the constructor, the lock should initially be open):

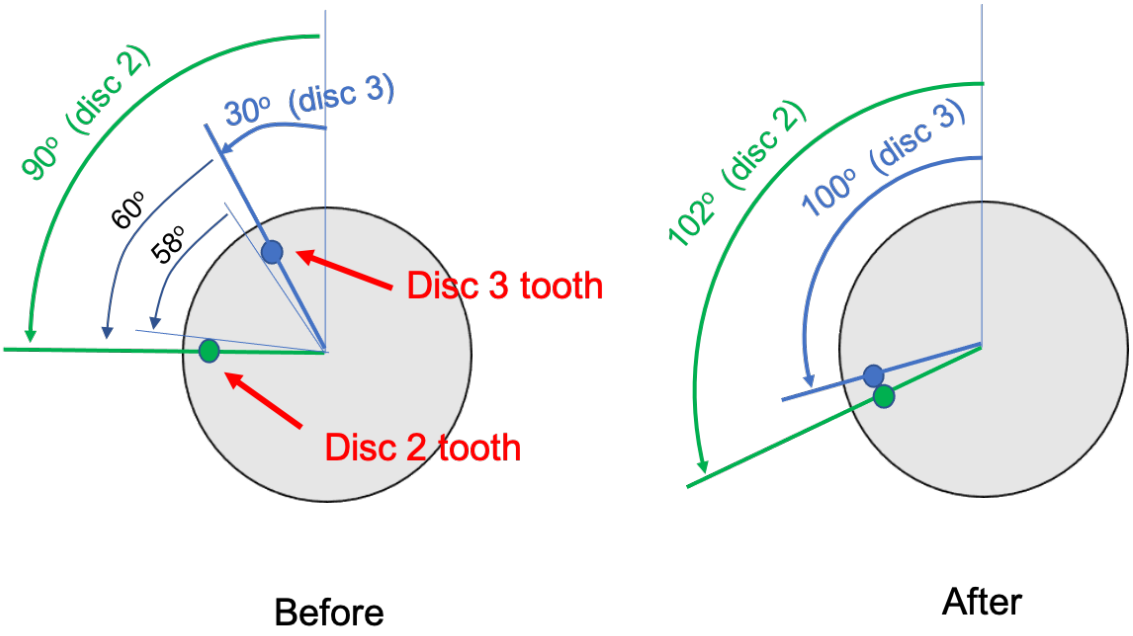
```
  System.out.println(p.isOpen());       // expected true
  System.out.println(p.isAligned());    // expected false
  p.close();
  System.out.println(p.isOpen());       // expected false
  p.open();                             // does nothing; it's locked
  System.out.println(p.isOpen());       // expected false
  p.setPositions(6, 22, 30);
  printPositions(p);                    // expected 6, 22, 30
  System.out.println(p.isAligned());    // expected true

  // now we should be able to open it!
  p.open();
  System.out.println(p.isOpen());       // expected true
  System.out.println();
```

4. To start thinking about what happens when you rotate the dial, first think about just disc 3, the front one that is attached to the dial. You could start by just making sure you can get disc 3 to end up in the correct position:

```
p.setPositions(4, 2, 0);
p.turn(10);                                  // 10 degrees ccw
System.out.println(p.getDiscPosition(3)); // expected 10
p.turn(-100);                                // 100 degrees cw
System.out.println(p.getDiscPosition(3)); // expected 270
p.turn(800);
System.out.println(p.getDiscPosition(3)); // expected 350
System.out.println();
```

5. Now it gets interesting. How does disc 3 affect disc 2? Suppose disc 3 is at 30 degrees and disc 2 is at 90 degrees, and we rotate disc 3 counterclockwise (ccw) 70 degrees.

```
p.setPositions(0, 90, 30);
p.turn(70);
printPositions(p);  // expected 0 102 100
```



Here, (disc 2 position) – (disc 3 position) = 90 – 30 = 60 degrees, the counterclockwise rotation from disc 3 to disc 2. There is also the tooth width to account for, so disc 3 can rotate 58 degrees before it starts pushing disc 2. We're rotating disc 3 a total of 70 degrees ccw, so disc 2 will be pushed 12 degrees ccw, ending up at position 102.

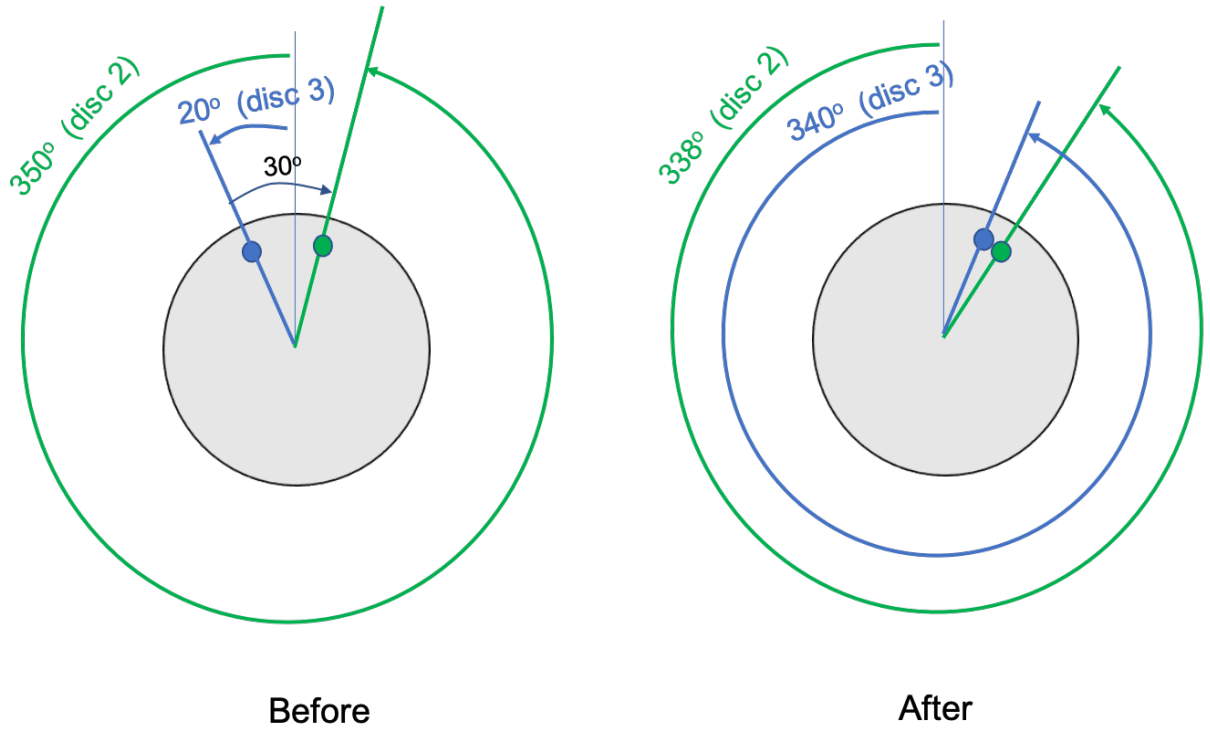
Maybe we should try one where the subtraction comes out negative:

```
p.setPositions(0, 20, 350);
p.turn(50);
printPositions(p);  // expected 0 42 40
```

In this case, (disc 2 position) – (disc 3 position) = -330 degrees, which normalizes to 30 degrees. With the tooth width, disc 3 can rotate 28 degrees ccw before affecting disc 2. We're rotating 50 degrees, so disc 2 is pushed 22 degrees.

6. Then try going clockwise:

```
p.setPositions(0, 350, 20);
p.turn(-40);
printPositions(p);  // expected 0 338 340
```



Here, (disc 2 position) – (disc 3 position) = 330 degrees, but we want the *complement* of this angle since we're rotating the opposite direction, which is 30 degrees. (Equivalently, just perform the subtraction in the opposite order and normalize.) Less the tooth width that's 28 degrees. Disc 3 is rotating a total of 40 degrees clockwise (cw), so disc 2 will move 12 degrees cw.
New disc 3 position = 20 – 40 = -20, which normalizes to 340.
New disc 2 position = 350 – 12 = 338.

Try another one:

```
p.setPositions(45, 40, 30);
p.turn(-400);
printPositions(p);  // expected 45 348 350
```

Here, (disc 2 position) – (disc 3 position) = 10 degrees, but we want the complement of this angle since we're rotating the opposite direction, which is 350 degrees; less the tooth width that's 348 degrees. Disc 1 is rotating 400 degrees cw, so disc 2 will be pushed 52 degrees.
New disc 3 position = 30 – 400 = -370, which normalizes to 350.
New disc 2 position = 40 – 52 = -12, which normalizes to 348.

Accounting for the effect of disc 2 on disc 1 is similar. In the test case above, for example, we can tell that the difference, (disc 1 position) – (disc 2 position) = 45 - 40 = 5 degrees, the complement of which is 355, less the tooth width is 353; but disc 2 is only rotating 52 degrees cw, so disc 1 is unaffected. If we instead had the same example but with disc 1 at 10 degrees:

```
p.setPositions(10, 40, 30);
p.turn(-400);
printPositions(p);  // expected 346 348 350
```

Now, (disc 1 position) – (disc 2 position) = -30 degrees, normalized to 330, but we want the complement which is 30, and allowing for the tooth width, we get 28 degrees cw from disc 2 to disc 1. Disc 2 is moving 52 degrees cw as in previous example, so disc 1 will be pushed 24 degrees cw. New disc 1 position is 10 – 24 = -14 or 346 degrees.

7. Once **`turn`** is implemented, it is easy to implement the methods **`turnLeftTo`** and **`turnRightTo`**. After that, you should be able to open the lock according the instructions on the package (recall we originally constructed a padlock with combination 10, 20, 30):

```
p.turn(-720);  // spin it twice clockwise
p.turnRightTo(10);
p.turn(360 - Padlock.TOOTH);  // counterclockwise a full revolution
p.turnLeftTo(20);
p.turnRightTo(30);
System.out.println(p.isAligned());  // expected true
p.open();
System.out.println(p.isOpen());     // hooray
```

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_hw2.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw2`**, which in turn contains one file, **`Padlock.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 2 submission link and verify that your submission was successful. 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 files **Padlock.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.