Homework 2

CSCIE-55

Fall 2019

Fred Evers

# Homework 2: Expanded Elevator Model w/ Polymorphism, Java Exceptions and more!

Due: Tuesday, October 1st 11:59 Eastern time

Grading: 15%

Last Modified: Wednesday, 17 Sept 201911:39:58 EDT

We are taking the first steps in the creation of a system that we will work on for most of the semester. It is a small world in which there is one building with seven floors, apartments on each floor, and residents in the apartments who ride elevators. This system will grow as we progress and will present us with many challenges and opportunities to use the features of Java.

Below is a UML class diagram of our project. [Zoom in. I hope this helps]
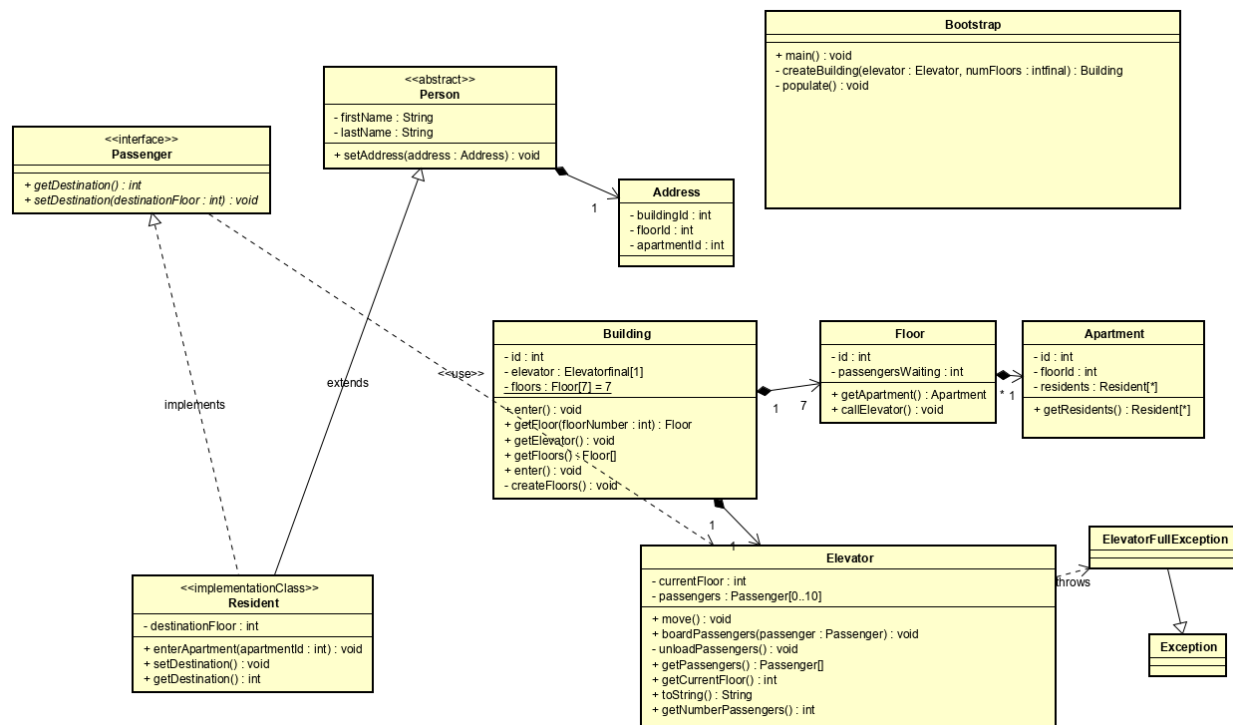
Specification of Requirements

[Note: Requirements are not suggestions, they are requirements. Your work should meet them strictly and completely. Do not supplement or modify them. Do not try to design a better Elevator, Building, Apartment. There is plenty of room for innovation while meeting the requirements. In this respect the assignments mimic real world software tasks.]

Your submission should include the following:

- An executable .jar file named <lastName>_<firstName>_hw2.jar
- A README.txt file describing how to run the program from the .jar file.
- JavaDoc for each class, placed in a directory inside the jar named 'docs'.
- Push your code into BitBucket and submit the final results on Canvas.
- **NOTE:** a new pom.xml file is added on 9/24. This pom will generate 3 jars, including one named building-1.0-SNAPSHOT-tests.jar That jar is the executable jar, and will be generated with a correct MANIFEST.MF file.

This homework extends Homework 1 in the following ways:

The classes should be put into the package specified. The assets downloaded from BitBucket will contain stubs for the classes you will implement. Do not modify the structure or the package names.

There are several new types of object [classes].
- Building
- Floor
- Apartment [*provided*]
- ElevatorFullException
- TooManyResidentsException [*provided*. **You can ignore this**.]
- Person [*provided*]
- Passenger [*provided*]
- Resident
- Address [*provided*]
- Bootstrap [*provided*. **You can ignore this**.]

Overview

The Elevator from last assignment will be refactored and reused. Be sure the package name matches the current specification.

The Elevator moves passengers between Floors. The Elevator and the Floors are part of a Building.

The Elevator will limit the number of passengers it can carry, throwing an ElevatorFullException if an attempt is made to board passengers past this limit.

Instead of writing a main(String[] args) method, your implementation will be tested by JUnit tests that are present in the src/test directory.

While it is recommended that you extend your Homework 1 code, that is not required, and you may prefer to start Homework 2 from scratch. Either approach is fine. The requirements given below are written with references to the Homework 1 requirements.

Tasks

Code organization

The Elevator, Floor and Building classes all go into the package **cscie55.hw2.impl** If you define any additional classes, they should also go into this package (and should not be declared public).

The Elevator class

The Elevator moves exactly as it did last time, dropping passengers at Floors, and to pick up new passengers. But now, if a passenger tries to board the Elevator when it is already loaded to capacity, an ElevatorFullException will be thrown. That should prevent the passenger from boarding. [It's up to the move() method to deal with the exception, rather that throwing it up the stack.] The Elevator will pick up the passengers that could not be accommodated when it next visits the Floor.
Package: cscie55.hw2.impl

Starting with Elevator from homework 1 or from scratch, add [or if reusing, verify they still work in this context] the following the **public** methods and fields* to the Elevator class:

- NEW: Modify the constructor for Elevator so that it takes a parameter as follows: Elevator(Floor[] passWaitingArray)

-

- int getCurrentFloor() : Return the Elevator's current floor number. I.e., this is the number of the floor reached by the last call to Elevator.move().

- void move(): this is the method you call to make the elevator move, as in Assignment 1. See "enum direction" below.

- int getNumberPassengers(): Return the number of passengers currently on the Elevator.

- void boardPassenger(int destinationFloorNumber) **throws ElevatorFullException**: Board a passenger who wants to ride to the indicated floor. Note that this method boards a single passenger and may throw an ElevatorFullException.

- void boardPassenger(Passenger passenger) throws **ElevatorFullException.** In this case, **you will create an overloaded method that takes a Passenger object** [interface provided] as a parameter. This method will extract its 'int destination' value from the Passenger object given in the parameter. This 'destination' is then used to add to the array-backed tracker by which you track passengers headed to the various floors, just as the method above does.

- Elevator.CAPACITY is a **final static** field that stores the number of passengers that the Elevator can accommodate. Set it to 10.

- private **enum direction {UP, DOWN}.** This construct is to be used to indicate the elevator's direction – i.e., 'state'. Use this in your move() method to determine when to reverse the direction of the elevator.

*[Note that the term field is used here as synonymous with the terms "instance variable" and "state variable".]

The Floor class

This is a new class, representing one of the floors that the Elevator can visit.
Package: **cscie55.hw2.impl**

It should have the following public methods:

- int getPassengersWaiting(): Returns the number of passengers on the Floor who are waiting for the Elevator.
- void callElevator() : Called when a passenger on the Floor wants to wait for the Elevator. Calling this should cause the Elevator to stop the next time it moves to the Floor. NB: For this

homework assume that passengers waiting for the Elevator on floors 2 and above should all be boarded as going to the first floor. (We'll drop this assumption in the next homework.)

- Floor(int id) This is the Floor constructor. The Floor constructor will be responsible for adding 4 Apartments to each floor. [The Apartment class is provided. See its's constructor for what you need to add to the body of the Floor constructor.

You will need to decide what fields the Floor class should have. Note that to meet the first two requirements a Floor object must retain a piece of state that records the number of passengers it has waiting for an Elevator.

The Building class

This is a new class, which keeps track of one Elevator and multiple Floors. It provides access to these objects for tests. Building has the following public methods and fields.
Package: *cscie55.hw2.impl*

- getElevator() : Returns the building's Elevator
- getFloor(int floorNumber): Returns the Floor object for the given floor number.
- *TOTAL_NUM_OF_FLOORS*: A **static final** field storing the number of floors in the building. Set the value to 7.
- *Floors*: a private array of type FLOOR to track the passengers as they come and go.
- Building(): The Building constructor creates an Elevator, and one floor for each floor number.
    - o NOTE: To permit communication between the Elevator and the floors of the building [floors contain an int value named passengersWaiting], we will need to modify the
- private createFloors(). This method should populate the floors array with the required nubmer of floor instances. See the constructor for Floor [or the BootstrapTest usage] to determine what is required to create Floor objects.

These specifications are requirements. As such, they specify things that must be done in any correct implementation. You are, of course, free to add other methods (as long as they are not public) and any fields (that should probably not be public).

The ElevatorFullException class
This needs to be a public class, extending java.lang.Exception. ElevatorFullException, like any other exception, must be thrown when something out of the ordinary happens. Because we do not want this exception to end execution of your program, some other code must catch ElevatorFullException and handle the situation, (i.e., that the elevator cannot board a passenger when it is full).

Package: cscie55.hw2.exception

In this case, Elevator.boardPassenger(int destinationFloorNumber) throws ElevatorFullException when the Elevator is already at capacity.

Elevator.boardPassenger(int destinationFloorNumber) is called when the Elevator stops on a Floor, some passengers leave, and the passengers waiting on that floor try to board. This code must catch the exception, e.g.

```
try {

   ...

   elevator.boardPassenger(...); // Method may throw ElevatorFullException

   // The passenger boarded successfully

   ...

} catch (ElevatorFullException efe) { // catch exception

    //  handle exception

   // The passenger was unable to board because the elevator is full.

   ...

}
```

The Resident class

This is a class that extends Person [provided] and implements the Passenger class. As such, it inherits all the fields and public methods from it's super class [Person]. You are responsible for creating methods that conform to the methods specified in the interface 'Passenger'. Create a constructor that uses the same parameters as the super class. It must call: super(firstName,lastName,address); on the first line of its constructor. If you feel you need any other additions to the constructor, they must follow the call to super().

Design Considerations and Gotchas

It should be clear from the requirements that there must be communication between Floor objects and Elevator objects. This is in order that the Elevator knows when there are passengers waiting to board when it arrives at a Floor and that the Elevator can tell its current Floor when it has boarded one of the waiting passengers. In other words, your model must maintain data integrity. There are different design approaches to achieve this. It's up to you which approach to take, so long as it works and so long as the code that implements it is easy to understand and well commented.

Array sizing and indexing present potentially fatal gotchas. If you create an array to represent data for 7 Floors and you size the array to be 7 cells, its 0th element will map to floor number 1 and its 6th element will correspond to floor # 7. When you want the data for the ith floor it will be in cell i-1. The key to avoid indexing errors is careful attention to preserve this mapping.

Further design considerations. Homework 2 is a big expansion of the simulation model. Structurally there are several new classes. In terms of data, the big change is the presence of passengers waiting on floors. The big change in behavior is that Elevator.move() now must take into account the need to board waiting passengers when it arrives at a Floor, that is, if there are any such waiting passengers, even if there is no one on board destined for the new current floor. And when it stops there it must board only as many passengers as it can accommodate, meaning that some passengers will be left waiting on the Floor, until the next time the Elevator arrives there.

How does a Floor acquire waiting passengers? That's what callElevator() does. Further, the Floor class must keep track of how many passengers are waiting, and be prepared to tell the Elevator when it arrives there. Notice this means Elevator and Floor must communicate.

As the Elevator boards a passenger waiting on a Floor, how does the Floor learn to decrement the count of waiting passengers? We will need a method by which an Elevator tells a Floor it has taken one of its waiting passengers. Does this method have to be public? The homework states that no further public methods may be added. You don't have to make a method public for other classes in the same package to have access: absent an explicit qualifier such as 'public', 'private', or 'protected', a member is 'package protected', i.e. accessible to every class in the same package.

Opening the BitBucket Project in your IDE

When a project is under Maven's build structure, a management file named 'pom.xml' contains the dependencies and configuration information required to resolve all dependencies. It also may have plugins that determine what happens when the project is packaged.
To open the assignment, click your IDE's File menu and select 'open'.

Navigate to the location of your homework_2 cloned project. Point to the pom.xml file, and click OK or open or whatever. This will bring in the entire project. In Intelli-J it looks like this:
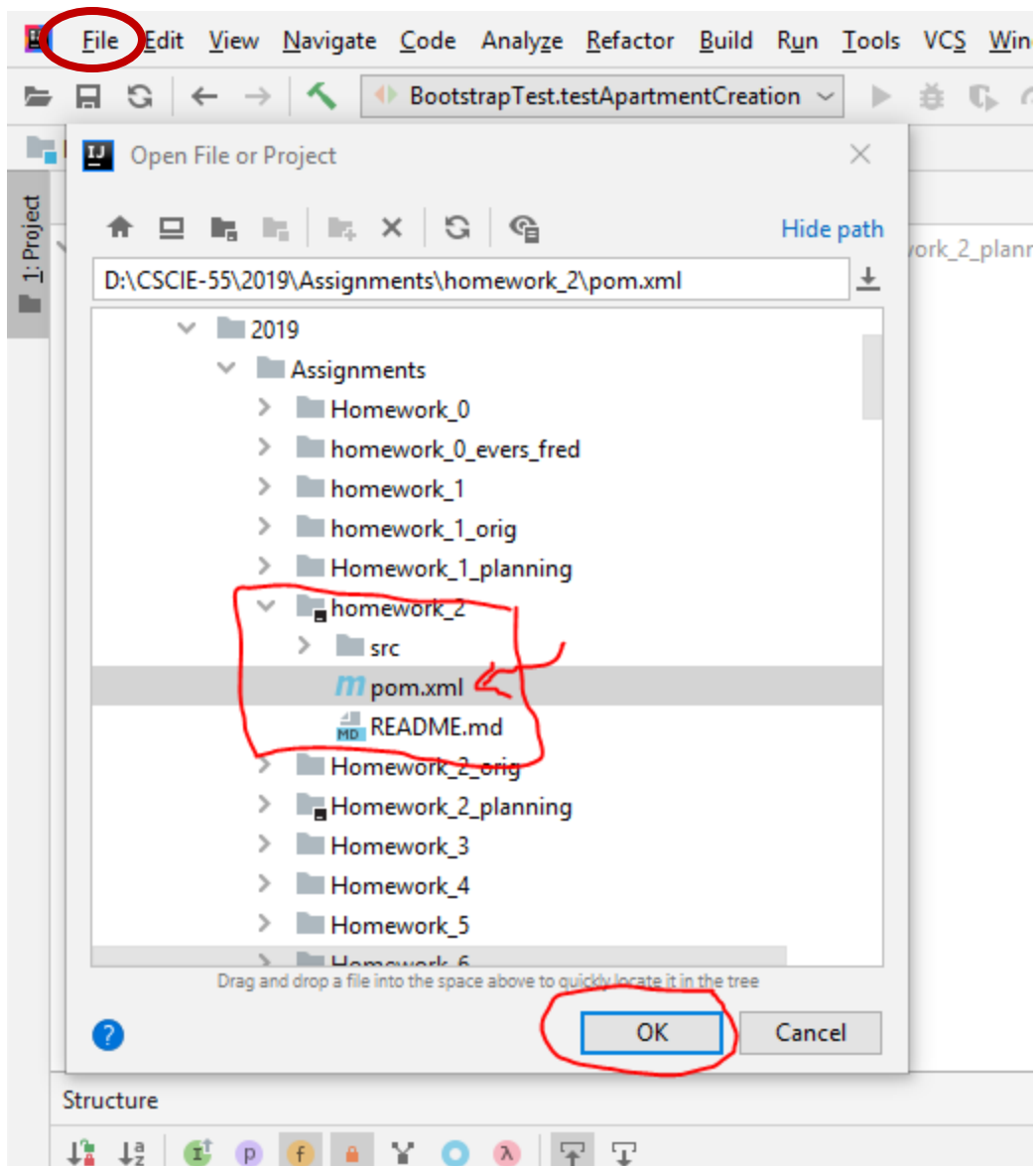
JavaDoc

We want you to document your code. Use JavaDoc comments as you work, applying the comments we discussed in class and in section. Document public and significant private methods. You can leave out getters and setters if you wish. A maven plugin will take case of packaging your documentation.

Testing

In Homework 1, you demonstrated that your code was working by writing a 'test harness' - a main(String[] args) method that created an Elevator, set up some passengers, and then called Elevator.move() and generated output to show that things were working properly.

A more common approach to testing is to write self-testing code. You write a test that sets up initial conditions, moves the Elevator a few times, and then examines the state of the Elevator to see that all is as expected. A program like this has no output if all works well. However, if any state is not as expected, this results in an assertion failure, which is printed, and that test is then reported as failing.

The most common self-testing framework for Java is JUnit.

As before, submit the output you get when you run your program. If all goes well, there should be minimal output, e.g.

   JUnit version 4.11

   ....

   Time: 0.006

   OK (4 tests)

If you don't see something like this, and there is output indicating tests failing and containing stack traces, then tests are failing. Figure out what's going wrong and keep debugging until you get a clean run. A good strategy is to comment out failing tests and run them one at a time until they pass.

Getting started with JUnit

For general background on JUnit see page "Unit Testing with Junit" linked to the assignment posting.

The first thing to do is to download the assignment code. [Note: **You may not modify the test code**. If it fails to compile, your implementation has violated the requirements. If any tests fail, there are errors in your code.] The class Java code where the tests are written is cscie55.hw2.BootstrapTest. Note that the package is cscie55.hw2.

We learned about 'classpath' in Assignment 1. The classpath is a like a url: it is a location pointer indicating where the needed resources can be located on the computer that hosts the program [and the resources needed]

We are now running our classes under the Maven build system. When dependent libraries are listed in the <dependencies> element of the pom.xml file, Maven will automatically download the libraries your program needs, put them into a directory named .m2/repositories under your home directory, and will resolve the classpath automatically.

Unlike our problems with classpaths in HW1, you can now run tests from inside your IDE because modern IDEs are Maven-aware. Maven understands the structure of the project's code. That problem is solved! Now you can do everything you need from the IDE and from the command line.

Maven from the Command line.

Navigate to the root of your project, I assume that is where you cloned hw2:
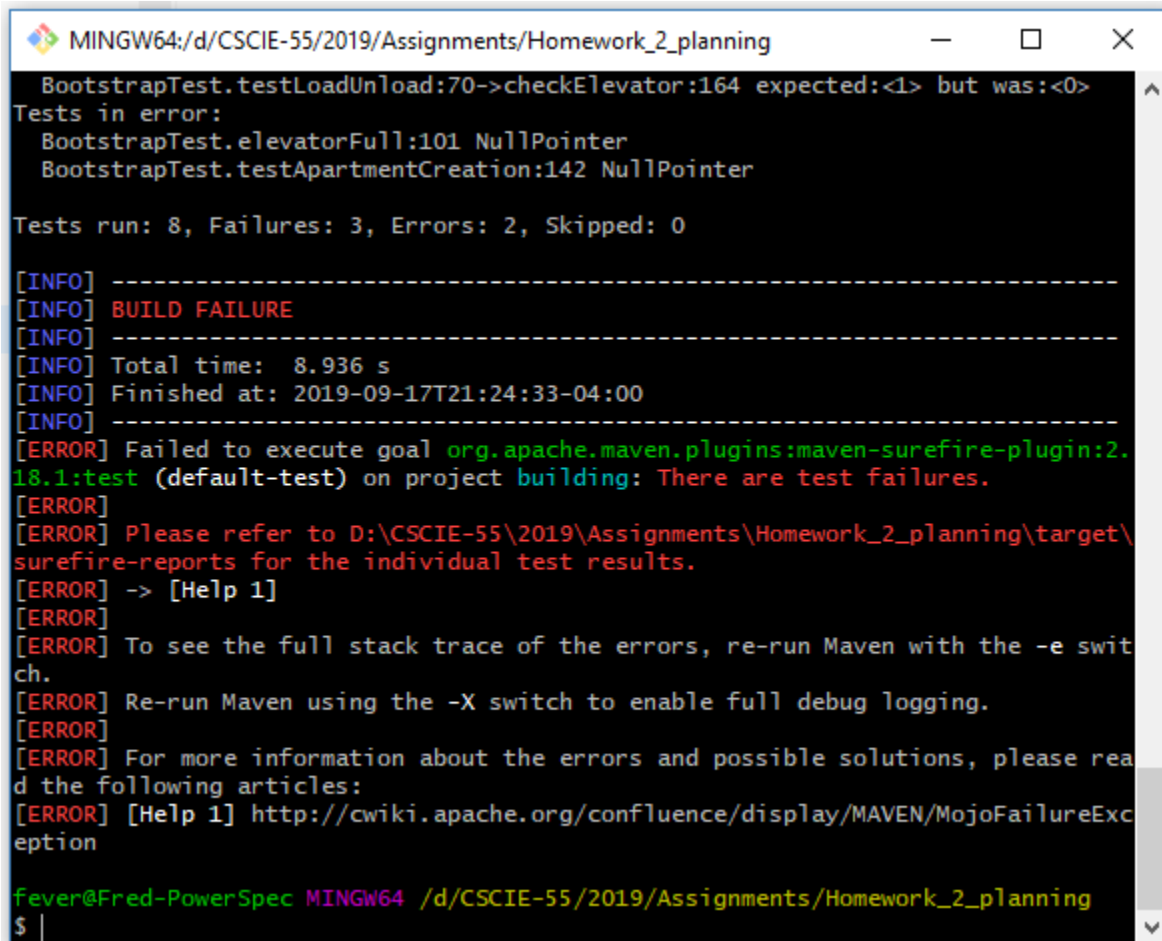
Assignments/homework_2_<lastname>_<firstname>

To compile your code, do: **mvn clean compile**

You can run the tests on the command line [terminal or Git bash shell] by navigating to the top-level directory of your project ['assignment_2'] and doing: **mvn clean test**

- The 'clean' directive removes old compiled versions of your class files.
- 'test' looks for Unit tests under the 'test' branch of your code and execute any it finds.

You will see output like this:

```
MINGW64:/d/CSCIE-55/2019/Assignments/Homework_2_planning          —     □     ✕

  BootstrapTest.testLoadUnload:70->checkElevator:164 expected:<1> but was:<0>
Tests in error:
  BootstrapTest.elevatorFull:101 NullPointer
  BootstrapTest.testApartmentCreation:142 NullPointer

Tests run: 8, Failures: 3, Errors: 2, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  8.936 s
[INFO] Finished at: 2019-09-17T21:24:33-04:00
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.
18.1:test (default-test) on project building: There are test failures.
[ERROR]
[ERROR] Please refer to D:\CSCIE-55\2019\Assignments\Homework_2_planning\target\
surefire-reports for the individual test results.
[ERROR] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e swit
ch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please rea
d the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureExc
eption

fever@Fred-PowerSpec MINGW64 /d/CSCIE-55/2019/Assignments/Homework_2_planning
$ |
```

You can create jar files by doing: **mvn clean package**

Because you are running a junit test, the class you run is **org.junit.runner.JUnitCore. That is, the main method is in class JUnitCore,** which is part of the Junit framework. There is one argument when running this application from the command line, the name of the test: **cscie55.hw2.BootstrapTest**

Packaging

Since your project is running under Maven, plugins can be added you your pom.xml file that control what happened when you run **_mvn clean package_**

Plugins have been added that ensure that your source code, tests and runtime jars are created.

NEW: After replacing the pom.xml file with the 9/25 version, you will have an executable jar by virtue of Maven's automagic [with help from Probal]

To run, simply cd into the 'target' directory where all the jars are, and do:

java -jar building-1.0-SNAPSHOT-tests.jar cscie55.hw2.BootstrapTest

You will see output in the directory that looks like this:

```
Results :

Tests run: 7, Failures: 0, Errors: 0, Skipped: 0

[INFO] Copying hamcrest-core-1.3.jar to D:\CSCIE-55\2019\Assignments\Homework_2_
planning\target\lib\hamcrest-core-1.3.jar
[INFO]
[INFO] >>> maven-source-plugin:3.0.1:jar (attach-sources) > generate-sources @ b
uilding >>>
[INFO]
[INFO] <<< maven-source-plugin:3.0.1:jar (attach-sources) < generate-sources @ b
uilding <<<
[INFO]
[INFO]
[INFO] --- maven-source-plugin:3.0.1:jar (attach-sources) @ building ---
[INFO] Building jar: D:\CSCIE-55\2019\Assignments\Homework_2_planning\target\bui
lding-1.0-SNAPSHOT-sources.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  10.709 s
[INFO] Finished at: 2019-09-24T19:50:40-04:00
[INFO] ------------------------------------------------------------------------

fever@Fred-PowerSpec MINGW64 /d/CSCIE-55/2019/Assignments/Homework_2_planning
$ cd target/

fever@Fred-PowerSpec MINGW64 /d/CSCIE-55/2019/Assignments/Homework_2_planning/ta
rget
$ ls
building-1.0-SNAPSHOT.jar          generated-sources/        maven-status/
building-1.0-SNAPSHOT-sources.jar  generated-test-sources/   surefire-reports/
building-1.0-SNAPSHOT-tests.jar    lib/                      test-classes/
classes/                           maven-archiver/

fever@Fred-PowerSpec MINGW64 /d/CSCIE-55/2019/Assignments/Homework_2_planning/ta
rget
$ java -jar building-1.0-SNAPSHOT-tests.jar cscie55.hw2.BootstrapTest
JUnit version 4.11
.......
Time: 0.006

OK (7 tests)     <---

fever@Fred-PowerSpec MINGW64 /d/CSCIE-55/2019/Assignments/Homework_2_planning/ta
rget
$ |
```

Submission

To submit, load each of these plus a README file to Canvas.

Good luck! And Enjoy!