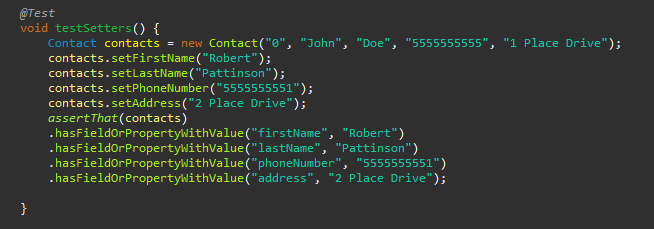
My approach to unit testing was to go class by class and make sure they were first implemented fully. I made sure that everything was completed before testing anything first just to make sure everything was there and I could just go through all the tests at once. I had wanted to ensure that the functions behaved properly before bothering with any tests even though the testing would have shown me any errors in my code anyways. For each test, I started with first making sure that every constructor worked, so I started with the Contact test and went through the ID, first and last name, phone number and address by using assertions to compare each variable with the values I set when calling the creating the object using the constructor.

The constructors, getters and setters are mostly comparing values with the initial variable, but for functions such as testIDLength, testFirstNameLength and other tests for the software requirements, I used the fail() function to fail a test if a certain condition was met. For instance, one of the requirements for the contact class was to make sure that the first name was less than or equal to 10 characters long. To make sure that it was 10 or under, I had just written an if statement that checked whether or not the name was longer than 10 and if it was, the test would fail with the error message saying that it was greater than 10. Just in case, I had made sure to truncate the string if it were greater than 10 regardless, so it would never fail if a regular user were to enter their name in. I used this method of testing multiple times all throughout the rest of the tests for the other classes and it seemed to always work well.

Overall, my JUnit tests could have covered more. Certain tests I found redundant, such as testing for both the first name and the last name in the Contact class since they essentially have the same functionality, so I didn’t do as many tests in the Appointment class. I prioritized ensuring the constructors worked properly since I was having a lot of errors with them at first and once I verified that a couple getters and setters also worked as well as a major function such as addAppointment() and deleteAppointment() I called it there. There are a few more functions that could be tested, however because of my other tests within these classes, I have a high degree of confidence that it will be okay, however, there is definitely room for improvement.

I enjoyed writing the JUnit tests since I had never done anything like this before. It was efficient once everything was set up and I was able to get into the motion of things, but I still think I would prefer to test functions the old fashioned way since it doesn’t require creating a whole new set of classes just for testing. It is certainly a great way to find bugs in your code, but simply printing to console whenever something goes wrong works just as well in my opinion. At least for minor bugs and errors that we’ve seen in this project. Perhaps for large scale projects where various issues could occur, it might be beneficial to do JUnit testing.

To ensure that my code was technically sound, I made sure that during my JUnit testing, everything ran without error. This was a surefire way to see if there were any glaring issues since in all my tests. I also made sure to constantly change variables around to see if anything was slipping by my tests just in case.  


In this bit of code, when testing the setter functions, I made sure that the constructor was filled with nonsense before setting, then testing the setter functions after. Since I had already tested the constructor and made sure that it worked properly just before this, I was sure that it was working and that my object was filled with the generic name John Doe and other basic information. This exact method of testing immediately proved that my setters worked and when they initially returned with errors, I knew exactly how to fix them.

Some software testing techniques that I used where using multiple different inputs when testing. If I had only tested one set of inputs in the constructor for example, “John” “Doe” “5555555555” and “1 Place Drive” and it passed, then I might not know what happened if I entered in a name that was more than 10 characters long, and ID that was negative, or an address that was more than 10 characters long etc. Testing as many different inputs as you can will lead to a better product and you can sleep well at night knowing you didn’t miss anything that would lead to a user crashing your program.

There are different practical uses for different software development projects and situations. In my case, when I program it is typically programming the front end of a website and making sure that everything you click works. These websites that I program aren’t for massive billion dollar companies and they aren’t very complex. I can get away with outputting errors to the console whenever something doesn’t work in order to find a bug and it will be a fine way of debugging considering I’ve become efficient at it. But in cases where you are working for that billion dollar company with a massive backend and thousands upon thousands of lines of code, you really do need to take it seriously and use JUnit testing or something similar because it would just be far too complicated to search for each bug manually by just posting to the console.

I wasn’t too keen on debugging in ways that were different to how I’ve been doing it for so long, and I wasn’t happy with having to set up JUnit testing which seemed a little bit too complicated just for testing code, but in the end, I learned how useful it actually is and once it actually was all set up, I was moving very fast and testing everything that I needed to pretty efficiently. This definitely changed my perspective on the importance of testing your code properly. I learned a lot with this project and I definitely will be using JUnit testing in my work down the line depending on the complexity of my projects.

I just explained how I was very against using JUnit testing or rather just testing in a different way than I was used to, and I had a lot of biases with the code that I wrote. I have been writing constructors, setters and getters, and just general programs like this one for so long now, I was so certain that it would be a walk in the park and this project would be a waste of energy. But once using JUnit testing, I quickly realized that I actually had made a lot of silly errors in my code just because I was too confident and perhaps cocky just because it felt like something I had done over and over again. This showed me that I certainly do have biases in my code since I overlooked so much just because I felt like it surely couldn’t have been wrong as its “just a constructor” or “just a setter function”. It’s so incredibly important to look at your own code with the mindset of it being the first time you’ve laid eyes on it. I think that whenever I look at other people’s code, I am quick to find any flaws if they exist, but with my own code that I wrote and have been staring at for hours, it can sometimes feel like the bugs are invisible. These biases are what lead to cutting corners and when you cut corners you could lead your team down a really bad path especially if what you’re programming for is something that deals with consumer data or anything at all sensitive. Brushing off your testing really could be the difference between having a good product and your company ending up in the news the next day.