Alec Maughan Assignment Report 19513869

Menu implementation:

My menu options are displayed all at once in the main menu, which is shown when first running the program, and a limited menu is shown at the end of every utility. Both use character input. The menu uses colours to highlight the letters that represent each menu option so it is easy to learn which letter does what, making menu navigation fast and easy to remember. My input for the menu uses a do-while loop as it will always trigger once. The loop condition is a boolean which is set to true if the input is invalid (From empty input or wrong character) and set to false if the input is valid. These checks are done using a simple if-then-else and a case statement. The character is changed to uppercase and only the first character of the input string is checked so a user can input “view ships” and it will still take them to the View Ships utility. Once the user reaches the end of a utility, they are prompted with the same menu input and they are given a small list of suggested menu options, including an option to view the full menu if needed. For example, when a user finishes loading a file, they have the option to Load another file, View ships, Main menu, or Exit program. In these limited menus all menu inputs still work, so a utility not listed can still be accessed. This is to reduce visual clutter without decreasing usability.

Data validation:

For data validation, inside ShipClass, EngineClass, SubClass and JetClass there are validation submodules that return a boolean. These are used in the setters to validate syntax specific to the class field, while other invalid imports such as empty strings or wrong datatype are handled inside the setters so that error messages can be specific to the problem to make the program easier to use. All setters accept a string import so that the methods that call the setters do not have to worry about errors in parsing. (Eg. setDepth has a different error message for empty input, wrong datatype and out of range). All user input is taken as a string to reduce the amount of exception handling done at input. By taking a String the user can also be shown exactly what they input in an error message, which cannot be done otherwise. This also means that empty inputs can be checked, which cannot be done when inputting other data types directly. My default and copy constructors don’t need any validation but I used setters in my alternates to increase cohesion.

Implementation of functionality in different classes:

I had all of View Ships in my User Interface because I had implemented a getter in my ship storage for a ship at an index in the storage array. This combined with a getter for my current no. of stored ships meant I could have a super basic loop in my UI to print all ships currently stored without having a view ships method in ship storage, reducing coupling. For ship creation from user input I created default objects and used their setters, this way I could get very specific error messages from the setters and still allow the user to keep entering values. The only part of this implemented in ship storage is the use of the addShip method, which means coupling is low. File loading and saving is mainly done in the file manager class, with the UI providing the fileName and outputting the returned message/s. This way class responsibility is maintained. Destination checker uses the UI to get the distance integer and uses the methods in SubClass and JetClass to calculate the hours. I was able to do this without a custom method in ship storage because of the get ship accessors. This also maintains class responsibility. Find duplicates is mainly implemented in ship storage, it loops through the arrays and checks different indexes against each other using equals, it also checks that a duplicate ship has not already been found before adding it. The method exports an array of ships that are then output in the UI, which maintains class responsibility.

Inheritance/Downcasting:
Inheritance complicated my ship creation from both file and user input. I have seperate parts of code for setting serial, year and engine for both submarines and jets. We weren't taught downcasting so I was not confident in using it to simplify these methods down so the end result is likely not as cohesive as it could be. That's about it in terms of inheritance and downcasting though.

Challenges faced:
In terms of challenges I faced I didn't have any major issues. Getting file reading to work is always a bit of a pain but once that was sorted it was all good. Forcing myself to make the program nice to look at and to also have super descriptive error messages for everything did add a lot of work for myself but it was fun and i'm happy with the end result.