REPORT for ShipManager

I decided to implement a sub-menu for adding a ship manually as it allows the user to choose what type of ship they would like to add, which leads to the variables being asked to be specifically tailored to what they inputted. My menu structures use a do-while loop structure as it allows the menu system to pass through at least once, and then the user can exit when they please. I chose to have a case validation to check if the specified number was in range for the menu options, and I also had a try-catch that made sure an integer was inputted into the system. I did not separate the validation submodule as I didn't want to have the logic be repeated twice and when I was creating the menu system creating new menu options was easier to implement. I did not consider using strings or characters in my menu system as I found it easier to validate they were correctly inputted.

I felt like it was unnecessary to have to double up on validating variables within both user interface and each of the 3 classes. This approach leads to the problem that you enter all the values in before the program telling you that something is wrong and leads you to re-enter all of the values again. Before sending the values to each of the classes I did a try catch to make sure the scanner picked up the right type of variables, so the 3 classes did not have to deal with checking type, rather just checking the data was within the bounds they needed to be. I considered putting validation in both and I did originally but I found it messy, despite it having the benefit of allowing the program to tell the user that a value is not within bounds straight away.

I decided that any methods dealing with creating, changing, validating or outputting actual data of the ship objects belonged inside the container class whilst all the other methods that had equations such as calc travel were placed outside as they could easily rely on an accessor to gather data. It meant that the ship storage class dealt with all of the equations and for loops whilst utilising the methods within the ship, submarine and fighter jet class to grab values and also call methods that would output the objects to strings. I did not consider including more of the methods from other classes into these 3 as I wanted to keep it streamlined as to what their jobs are.

I did not have problems with inheritance in my design, it greatly simplified my design. I originally had two ways to add ships and two ways to process ships which meant a lot of duplicate code everywhere. When I started to implement inheritance it meant getting rid of duplicate code and going from having two methods for everything to one that managed them both. It was also beneficial to get rid of the common code within Submarine and Fighter Jet with inputting, exporting and validating duplicate class fields.

I have implemented down casting within my equals method for submarine, fighter jet, engine and ship classes because we want to check if the passed object is an instance of a ship, engine, submarine or fighter jet object. Down casting is an easy and simple way of checking if an object is an instance of another which is useful for the equals method when we need to check if an object is a submarine object or fighter jet method.

One of the biggest challenges I had in my design was with the validation of data inputted by the user. This was because I decided to have data validation only be within the container classes. It meant I had to build everything around the container classes checking data and meant that as a result, the user would not a great experience as they won't be able to tell if a value they input is a valid value and cannot easily get great feedback from. Another challenge I had was implementing the adding ships from file method which was complicated as I had too many methods that did little things when I should've had larger ones that dealt with processing the file overall, processing a line and then dealing with adding and parsing that into the actual ship objects.