1 •

I chose to validate my data as soon as they were entered in from the user. This meant that whenever I had to create an object from user-inputted values, it would ensure that the objects created would be valid and it would force the user to input valid data only. I forced valid input by placing do-while loops at my user input stage, which means that it will loops until the validation method (which returns a Boolean) returns a value indicating that the user input is valid. This approach was repeated for all class-fields/input. This validation method works irrespective of how the user adds an alliance through the menu; whether it be manually inputting values to add an Alliance (which integrates Kingdom class' addAlliance() method, or through loading a file (which integrates FileManager's reading and processing methods). In both cases, the data is validated before it is passed onwards to either the Kingdom class or the FileManager class.

This validation is performed in UserInterface itself. My UserInterface validation methods were Boolean methods that returned TRUE or FALSE based off of the conditions within the method. They check whether the inputted data abide by the limits set by the assignment specification, whether it be limiting what values a certain input can be, or what type of input can be entered i.e. string, double or integer.

I think my approach of validating in UserInterface is efficient, however it could have improved and made even more efficient if I had integrated the idea of code re-use better and made three submodules that catered for each respective data type input i.e. double, integer and string; instead of having multiple validation submodules for similar data-types such as validateBannermen and validateArchers. This would also mean that I wouldn't need to put an excessive amount of try-catch statements in each submodule, that catch the same type of exceptions. I did avoid repeating code furthermore in my FileManager by calling the existing validation methods in UserInterface, which promotes code re-use. This means that if I needed to edit any of my class-fields in House, Army or BannerClass, I would only need to change my validations in UserInterface only, instead of several places containing the same validation code.

2 •

By the end of worksheet 9, we had a lot of code repetition within our program. Many similar class-fields for both House and Army each, two arrays, methods in Kingdom essentially doing the same process; once for House-specific class-fields, then again for Army-specific class-fields. To prevent so much code repetition, we implemented the information common to both Houses and Army's in a Class of its own – AllianceClass – which consists of the *Alliance's* name, years and banner. That means that everything associated with those three class-fields was moved into its own class, and that class would extend to House and Army. This helped streamline our code a lot, as it meant that each class was only concerned with its own class-fields, specific to itself.

All user interactions were dealt with in one class – UserInterface – meaning all inputs and outputs were channelled through one class. This helps all classes simply take in validated inputs and do any necessary processing they wish to do, instead of having to take in input

in 5 different classes, following where they're coming from and going to, and making sure they're validated all throughout the program.

3 •

I felt as if Inheritance made things a lot easier for my understanding of how the program works and it made it much less hectic dealing with one less array. I learnt about good coding practices at the end of high school and earlier this year, so for me to understand having so much code repetition made no sense to me and it seemed impractical. In hindsight, I can see that it helps us understand and get a good grasp of what efficient code means, and what it looks like (and vice versa). The refactoring in Worksheet 9 and especially 10 (Association and Inheritance) was helpful for my understanding of my program's design despite Inheritance being a harder concept to initially grasp; especially the concept of class responsibility.

4 •

My program didn't have any down casting present. I found ways to use abstract methods and overridden methods to make the program function as intended without breaking any rules in the assignment specification. My toString/toFileString methods are not abstract however are overridden in House and Army, therefore when AllianceClass' toString's are needed, the program will know what it's dealing with - an Alliance object filled with either House's properties, or an Army's properties.

5 •

I found it very challenging to form the structure of my Kingdom class (which is the container class of our program) i.e. creating classes based off the assignment specification (which is primarily what Worksheet 8 got us to do). I'm not sure if that's because of my inexperience with coding and/or whether it's because OOPD challenges you to think differently, but I really struggled with the idea of Object Orientation at first. I couldn't seem to understand how objects worked, as well as object arrays, so I was having a bad time constructing all those methods in Kingdom.

After the long hurdle of perfecting my Kingdom class, and making it error-free, my main challenges were just making sure I designed my other classes to work in unison with Kingdom properly. Once I got a sound understanding of how the program should be designed, and I was able to finish off Worksheet 8, Association and Inheritance were, much to my surprise, fairly easy for me to implement. I spent some time in extra labs asking tutors questions about Inheritance until I was happy with my understanding of it. This made it a lot easier for me to refactor everything to cater for Inheritance as it made a lot of sense to me and I could easily understand the relationship between subclasses and superclasses, and how they interact. Hence, after worksheet 8, I didn't face too many problems implementing those concepts into my program.