Lab 6: Abstract classes, user-input, and exception handling

This week, you are going to start making your code work in response to command line input as opposed to working in response to method calls while also working with the concepts of abstract classes and exception handling by catching bad input.

Most of your work this week will be in understanding the code you are presented with and showing that you can adapt it rather than writing a large amount of code yourself. We will not go into too much detail on how this code works because reading code to understand it is part of this week's exercise. In basic terms though, these three lines of code summarise it:

```
while (gameStillGoing)
{
      outputOptions();
      while (!userInput.hasNext());
      resolveInput(userInput.nextLine());
}
```

This code prints out the options that a player has based on the Game Mode, waits for them to type something at the keyboard and hit enter, they resolve their input and updates the game world.

Be aware that, for now, navigating the world has no effect except to slightly increase the chance of a random monster encounter. Adding in real maps and navigation will come later.

Note 1: This week, we will be marking submissions by hand by playing them, so getting the code to compile and run will be vital! We expect to mark almost all of these tests in game.

Note 2: Make sure you start this week using the new sample code available for Lab 6 in Moodle.

Task 1: Abstract classes

Let's start off by tidying up the class structure we have been using in out design of the Monster class inheritance system with the abstract keyword. The Monster class should not be able to be instantiated because we do not have any monsters that are just monsters – they all have a sub type of family and a specific breed subtype after that.

1. Make it so that the Monster class becomes abstract and the basicAttack, and specialAttack methods in it are abstract as well. Notice what, if anything, this break in your code.
2. Make the Family classes (Fire, Water, and Plant) abstract as well. If this breaks something in your code, think carefully about the correct way to fix it. If you have,

for example, implemented a normal monster last week, this change might be a problem for you.

Hints

- There aren't a huge number of benefits to using abstract classes in this way right now because we have done things backwards. When designing a system, we would normally make the abstract classes first and not be able to instantiate their subclasses until all the abstract methods they inherited were overridden.

Marks	Task	Name	Description
10%	*1_1*	*Abstract refactor of Monster*	We should no longer be able to create an object of type monster in your code
10%	*1_2*	*Abstract refactor of families*	We should no longer be able to create an object of type PlantFamily, WaterFamily or FireFamily in your code

Task 2: Making trainer work

We've provided you with a piece of starting code that can run a small game using the code created last week by printing out a set of options for a player to choose from exploring an open world based upon a set of different Modes. Look at the code in GameWorld and try to understand how it works.

1. Modify the startup code so that the printRoster(Boolean showAttacks) method will output all the monsters the trainer currently has, sorted by their readiness form most to least ready, with the conscious monsters showing the different attack options that they have available to them and no options showing for the unconscious monsters. If you have done the extra work in the previous weeks on making a nicely formatted set of monsters use that as a starting point.
    - We have set it up so that 1 is a basic attack and 2 is a special attack in the command line.

2. Modify the startup code to allow the player to use text input to select one of the three (or more) instances of Monster that you have implemented and give them a name. You can see in the starter code how this has begun to work but you will need to modify the code in two places:

    outputStartupChoices
    resolveStartup

    To add in all your choices. Notice how the structure of the code works with output of your choices depending on the mode the game is in and input resolution also depending on the mode. Also note how the game moves back to the EXPLORE state after you finish with most other states.

3. Modify the system so that it returns one of the three, or more, monsters you have implemented for a random monster encounter rather than only returning one type of monster.

Hints

- You are going to need to spend some time playing with the game – get the bare minimum done for task 2_1 then spend some time playing with the game to get an idea of how it works.
- Finding the code you need to modify for 2_3 is the main challenge. It's going to be a bit tricky to find but easy to modify when you do find it. Look carefully at how the code executes.
- Note that manage team does not work now – fixing it is your next job.

Marks	Task	Name	Description
15%	*2_1*	*printRoster*	Print out the roster ordered by readiness.
10%	*2_2*	*Increased selection of starter monsters*	Demonstrate you understand and can modify the game loop by altering the code in resolveStartup
10%	*2_3*	*Truly random monsters*	Analyse the code and find where you need to make a change to implement this expansion of functionality.

Task 3: Items and dealing with bad input through Exceptions

Look at the code in resolveManageTeam – you will find that, when you try to manage your team in game, you cannot get this code to work nicely! The game asks for the following:

Options:
-- Use an item on a monster: 'Monster-Number Item-Number'
-- rename a monster: 'Monster-Number New-Name
-- return to exploring the world: r

While an input of r will work OK, you will need to move your solution to last week's exercise (or one taken from the sample solution) into trainer to make the item selection work. In addition, now, if the input you get is not of the form "r" or two numbers you get an exception thrown!

1. Use a try and catch to stop badly formed inputs to the resolveManageTeam method from crashing the game. Start by looking at the resolveFightWiild method and try and make a similar, general exception catch in place that prints out a request for a better formed input.

2. Use Java's ability to handle multiple exceptions to detect when a NumberFormat Exception gets thrown and, instead of asking for more input, see if you can treat it as a request to change the name of a monster. Note

3. Move your item code over to the solution and try to make it so that the list of choices given to the player when they go into team management reflects the items available to them. Try and give the players a few random items when they start the game in the start-up mode when selecting monsters. It is your choice how this is achieved but let the player know what they have been given.

Hints:

- For task 3-2, be aware this isn't really the right way to do things – we don't normally make central logic hinge on an error. One reason why is that the error isn't always specific enough. There is a particular problem here that you might come across if you test your code well that will illustrate why!
- For task 3_3, pay attention to the order exceptions are handled in – refer to the lecture notes for more on this.

Marks	Task	Name	Description
10%	*3_1*	*Try and catch*	Implement simple exception handling
10%	*3_2*	*Number format exceptions*	Catch the specific exception and execute specific logic in response
25%	*3_3*	*Items*	Make a substantial change to the code to move over your own item code or import the sample solution code from last week's lab.

Bonus tasks

How could you add an extra state to the game? Could you add a way to have a team of your monsters fight an opposing trainer's team? Start by looking at the resolveExplore code and expand it with the chance of a random trainer encounter then build in your own game mode for trainer fights (and after trainer fight resolution as well).

If this code is working, you Can leave it in your submission this week.

Submission

This week's submission should be markable through playing the game. Make sure the features you have added are clearly presented to the player via the command line interface.