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

This week is all about starting to have more freedom with how you choose to tackle the problems we give you. You are going to start making your code work in response to command line input with exception handling to catch bad input, using abstract classes to tidy up the structure and expanding the game based on your own ideas. You can start to seriously play with the underlying code now – we're marking by playing your game so you can start to incorporate your own methods, data structures and classes into the code as it makes sense to you to do so.

We've made some more enhancements to help you use this code with user input:

- Coordinate and WorldEntity override their toString methods to help print details out.
- Encounter in WorldEntity prints details of each encounter that happens in a single line to help debugging.
- NPCs don't all move to cluster in one location on the map as we use randomLocation in world to generate a random goal for each of them.
- You have a very basic printWorld that capitalises the name of monsters that are awake and NPCs that have not reached their goal (i.e., things that should be moving). This is broken down with helper methods to shorten entity names and print a table out.
- Battles print out a detailed play by play of what happens in them to help debugging.
- resolveMove makes use of three helper methods – get NPC, terrain, and monsters here which returns an array list of all those things at the current location.

Task 1: Getting a Turn Structure Working and Managing Bad User Input

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. Look at the code in World and try to understand how it works. These lines of code summarise how the game loops:

```
    public void run() {
        int turnTimer = 1;
        while (!gameOver) {
            System.out.println("--------------------------------");
            System.out.println("TURN:" + turnTimer);
            System.out.println("--------------------------------");
            printWorld();
            adventurer.takeTurn();
            nonPlayerCharactersMove();
            monsterMove();
            turnTimer++;
        }
```

```
        System.out.println("Game Over!");
    }
```

If we look in the Adventurer class, we can also see their takeTurn method has been updated:

```
public void takeTurn() {
    printOptions();
    Scanner userInput = new Scanner(System.in);
    while (!userInput.hasNext());
    resolveTurn((Integer.parseInt(userInput.nextLine())));
}
```

This code will, once you get it working, print out the options that a player has currently got available to them (printOptions()), wait for them to type something at the keyboard and hit enter (while!userInput.hasNext())), then resolves their input by turning it into an Integer and calling the resolve turn method (resolveTurn). Once this is done, we will go back and have the NPCs in run to take their turn and then the monsters move as we had working last week. This will complete the turn and then we will loop back to the player again to get their next move.

Hint before you start:

- Glance over Tasks 2 and 3 before you make your choice here – some approaches can make it harder to extend your work to do Task 3.

You will need to:

1. Implement the printOptions method, thinking about what information you want to pass it as a parameter, to output the list of:
    a. All viable moves available to the player: North, East, South, and West moves of 1 that do not take the player outside of the world boundaries.
    b. All the NPCs at their current location: Modify your code so that players don't automatically 'encounter' an NPC when they move into a square with them, but instead must spend a turn interacting with them to get healed or buffed as an option on their list of choices.
        i. Think about how you can access this list – we have code that does this job for us already in the solution provided.

You need to consider how you can represent this list of options, especially given what task 2 is going to be. To allow the user to select an option by entering a number at the command line we use the java.util.Scanner[1] class. At the moment, we wait until the user types something, and hits enter, then we try and convert that to an integer and call resolveTurn with it.

2. The resolveTurn method needs to make the right thing happen – either moving the direction the player input or interacting with an NPC. You will need to think about

[1] Documentation at https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Scanner.html

how to do this – the code you already have does a lot of these jobs for you but not all of them.

a. How can you represent the choices?
b. How are you going to map the user input to a choice in a logical way?
c. How can you call the appropriate methods to resolve the choices?

Hints:

- This is starting to become 'real' Object Oriented Programming as you need to think about big OO design ideas and how to apply them without much guidance form us.
- A few ways you could solve this are to story an ArrayList of objects related to choices, to create an array of choices, to make a new Enum type for the different choices or to make a new Object to represent each choice option. None of these options will be wrong and each of them could be implemented in several different ways – think through a couple of them before you start to program one.

Exception Handling: Try entering something other than a number for the input when taking a turn and you should see that the game crashes out with an exception to the command line. Having the game crash each time you make a typo is not particularly satisfying so let's try and fix that now.

3. Enclose the call to parseInt in a try statement and then include a catch clause for NumberFormatException e. When you catch the exception, print out a message saying you can't make a message from the input (you can use e.getMessage() to get a short string version of the issue to make this printout more helpful).
    a. When you catch this exception, you should try and take the player turn again to give the user a chance at getting their input right – how can you do this simply?

Task 2: Abstract classes and Polymorphism

Let's tidy up the class structure we have been using in our design of the WorldEntity inheritance system with the abstract keyword. The WorldEntity, TravellingWorldEntity, Monster and NPC classes should not be able to be instantiated because we do not have any instances of these things in our world which are just these things – they all have a more sub type after that.

1. Make it so that the TravellingWorldEntitry, Monster, NPC and Terrain classes become abstract and make the attack method abstract as well.
2. Notice what this changes in your codes execution. For example, now in battle the call to attack will now use the specific attack of Adventurer without casting the battling object to an instance of adventurer because of polymorphism.

You will need to go through the code putting in implementations of the abstract methods – at this stage, you need to think quite carefully about how you want to handle the design of this.

1. If you put a concrete implementation of something into a class like Monster, then unless you specifically test for and cast each subclass of Monster, you will only get the functionality in the Monster class when you call a method.
2. If you make a method abstract in monster and put specific implementations of it into the subclasses, you will be able to call all the specific implementations of it from a reference to a Monster object. This is an example of the benefit of Polymorphism in Java where one name can refer to different, more specific instances. However, you will need to write more code and possibly duplicate a lot of effort.

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.
- There is NO RIGHT ANSWER at the moment for these questions – we simply want you to start to think about them because this is at the crux of good Object-Oriented design.
    - You will look more at many issues just like this after Christmas when we discuss, for example, the principle of composition over inheritance and the application of the Strategy Design pattern in Object Oriented Software Engineering.

Task 3: Finally Having Some Fun!

You need a ***total of at least three additional features*** for the game. You must add at least 1 of the features from 3.1, at least one from 3.2. Each of these features will require careful thought about how to modify the game world though none of them should require a total rewrite of the game world.

You should explain all the additional features when your game is started via a printout to the command line.

Task 3.1 Features

A. **Items**: allow the player to collect items as they travel through the world by exploring the locations that they are in. For example, a healing potion, magic armour that improves damage resistance or
B. **Cave systems (Hard):** Let the player enter caves – these will be smaller maps the player can explore where all the monsters in the map are awake and moving when they enter and there are more of them but there are more rewards as well.
C. **Multiple levels:** Let the player reach an endpoint that moves them, on to a new, harder map.

Task 3.2 Features

A. **Using Items (requires you did 3.1.A):** modify the option menu so players can use magic items like healing potions and equip armour.
B. **Interactive fights for the player (Hard)**: Give the player options in combat at the command line rather than just resolving the fight by who drops to 0 health first.
C. **Quests with NPCs**: Instead of getting a boon when then player first encounter an NPC, the NPC should give the player a mini quest like 'slay a specific monster' or find a specific item' they need to complete then return to the NPC to claim their reward.

Task 3.3 Features

You can choose one item from this list to add to your game or you may take on an extra challenge from 3.1 and 3.2. You can also add your own features you make up – just ask a tutor about the idea first to ensure it is appropriate.

A. NPC conversations via the command line: Let the player have conversations with the NPCs in the world.
B. Inventory (requires 3.1.A): Give the players a way to equip specific items and only carry a certain amount of them.
C. Game balancing: Modify the game world to make better use of the various features it has. Add more monsters or NPCs as you see fit.
D. Monster Expansion: Add in different types of monsters with different features – maybe they move faster, maybe they hide form the player on the game map.
E. **Terrain Expansion**: Make use of the volcanoes and mountains in world generation.
F. Game Over: Get the game to finish and summarise the adventurers run when they drop to 0 health.
G. Game fog: Instead of printing out a representation of the whole world, limit what the player can see based on where they are.

Submission

Submit all .java files via Moodle.

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.