CPSC 219: Introduction to Computer Science for Multidisciplinary Studies II

Assignment 2: Object-Oriented Java, File I/O, Packages, Objects, Enum, Inheritance, Polymorphism

Weight: 10%

Collaboration

Discussing the assignment requirements with others is a reasonable thing to do and an excellent way to learn. However, the work you hand in must ultimately be your work. This is essential for you to benefit from the learning experience and for the instructors and TAs to grade you fairly. Handing in work that is not your original work but is represented as such is plagiarism and academic misconduct. Penalties for academic misconduct are outlined in the university calendar.

Here are some tips to avoid plagiarism in your programming assignments.

1. Cite all sources of code you hand in that are not your original work. You can put the citations into comments in your program. For example, if you find and use code found on a website, include a comment that says, for example:

   ```
   # The following code is from
   https://www.quackit.com/python/tutorial/python_hello_world.cfm.
   ```

   Use the complete URL so that the marker can check the source.

2. A tool like chat-GPT can be used to improve small code blocks. For example, five lines of code. If you get help from code assistance like Chat-GPT, you should comment above the block of code you requested assistance on debugging or improving and cite the tool used to get that suggestion. Using a tool like chat-GPT to write the majority of your assignment requirements will be treated as plagiarism if found without citation, and with citation, it will be treated as 0 for the component the student did not complete. Code improvement of short length will get credit if commented/cited properly.
3. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. **However, you may still get a low grade if you submit code not primarily developed by yourself. Cited material should never be used to complete core assignment specifications. Before submitting, you can and should verify any code you are concerned about with your instructor/TA.**
4. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code, it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you exchange code with another student, write code while discussing it with a fellow student, or copy code from another person's screen, this code is not yours.
5. **Collaborative coding is strictly prohibited**. **Your assignment submission must be strictly your code**. Discussing anything beyond assignment requirements and ideas is a strictly forbidden form of collaboration. This includes sharing code, discussing the code itself, or modelling code after another student's algorithm. **You can not use (even with citation) another student's code.**
6. Making your code available, even passively, for others to copy or potentially copy is also plagiarism.

7. We will look for plagiarism in all code submissions, possibly using automated software designed for the task. For example, see Measures of Software Similarity (MOSS - https://theory.stanford.edu/~aiken/moss/).
8. Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor for help rather than plagiarizing. A common penalty is an F on a plagiarized assignment.

Late Policy

Assignments will be accepted up until 48 hours late with a penalty. From $0 < hours \leq 24$ will result in 10% penalty. From $24 < hours \leq 48$ will result in a 20% penalty.

Goal

Writing an Object-Oriented program in **Java**. Continue to use **Git** version control properly to store this project. Perform unit testing via **JUnit** to establish the correctness of portions of the assignment code created.

Technology

Java 16, Git, JUnit 5

Submission Instructions

You must submit your assignment electronically using **Gitlab** and **D2L**. Use the Assignment 2 dropbox in **D2L** for a final codebase electronic submission. You will also share a link to your **GitLab** codebase with your TA in that D2L submission. In **D2L**, you can submit multiple times over the top of a previous submission. Do not wait until the last minute to attempt to submit. You are responsible if you attempt this and time runs out. Your assignment must be completed in **Java** (not Kotlin or others) and be executable with **Java version 20.** You must use **Gitlab** hosted at **https://csgit.ucalgary.ca/** (not GitHub or another Git host). You must use **JUnit 5** and not other unit-testing libraries.

Description

You will complete a Java program, which is an MonstersVsHeroes simulation game. The game is simple relative to modern fantasy-world video games and will have no graphical portions or any real degree of user input. This game will load a file that describes the 2D grid world. This loading process will create Monsters and Heroes based on their location and properties given in the file. Then, the program will run a simulation where the monsters/heroes will move and attack each other without user control. The program will end when the user requests it to end or when all Monsters, or all Heroes, are no longer alive.

The goal of this assignment is to expose students to an Object-Oriented (OO) program. You will be provided with an existing framework of code already in an OO format. This means concepts of the game have been broken off from one single procedural file and stored (encapsulated) in separate files according to their purpose. Some of these files will be fully complete for students,

some will have parts that need completion, and others will be mostly up to the student to fully complete. There will be a variety of OO concepts visible throughout the code.

The code will be stored in packages.

Some code is used to **start the program**, like **Main**.

Some code will be built as **procedural helper classes** like **Logger/Reader**.

Other code will be **Enum** types like **Direction/WeaponType/Symbol** and internal **State** enumerations.

Some code will be a **regular object type** like **World**.

Some code will be **abstract,** like **Entity** and other code will **extend** that abstract code, like **Wall/Hero/Monster,** using polymorphism.

This variety of OO code will help students understand a variety of OO concepts, including encapsulation, polymorphism, enumerations, abstract, and static. Students will also get a chance to complete File I/O code.

Do not change the existing portions of the code. Complete functions only where indicated and add functions as specifically requested. You can add helper sub-functions if necessary. YOU SHOULD NOT IMPORT ANY OTHER LIBRARIES BEYOND java.io/java.util FOR THIS ASSIGNMENT, and you will get no credit for code copied from other places (cited code will get a grade of 0 as you are required to complete the required functions yourself).

Git Requirement: For this assignment, we will have the requirement that you upload your code to the university **csgit.ucalgary.ca** hosting site as a **private** repository shared with your TA (in addition to submitting it via D2L Dropbox). The minimum expectation for Git usage is that when you submit your code, the TA can go and view it in Gitlab and see that you've made multiple commits as you edited it before the submission deadline to D2L. To use **csgit.ucalgary.ca,** you will need a functional **UCIT account** username/password.

To Run the Game:

java Main world.txt log.txt 12345

> **world.txt** -> world file in a format that will be described later
>
> **log.txt** -> file to output logging of game progress to
>
> **12345** -> a seed to set random number generator for consistent execution (if you change this, then the program will run differently when it comes to random-based choices)

Example **world.txt** file

```
3
3
0,0,MONSTER,M,10,S
0,1
0,2
1,0
1,1
1,2
2,0
2,1
2,2,HERO,H,10,3,1
```

3 rows x 3 columns grid. First entry is rows, second is columns.

Monster is at (0,0) location with symbol M, health 10, and weapon type of (S)word. Other WeaponType options are (C)lub and (A)xe. Sword has attack strength of 4, Axe 3, and Club 2.(All Monsters have a static armour strength of 2 that is not entered via the file.)

Hero is at (2,2) location with symbol H, health 10, attack strength 3, armour strength 1.

When loaded in the game, this would look like the following (note, this is a 3x3 world in the file, but we are drawing it for viewing with external walls outside the valid array indices as #):

```
#####
#M..#
#...#
#..H#
#####
```

Where # are Walls. M is a Monster, and H is a Hero.

Program Coding Requirement:

This is a recommended order of completion, but each class can be completed/tested in whatever order you desire.

In World.java

You must complete **gameString()** (and associated **worldString()**)

public String gameString()

is a **World** method that changes a **World** into **String** for output. An example **String** looks like the following for the example input file given above.

```
#####
#M..#
#...#
#..H#
#####
NAME     S    H    STATE    INFO
Mons(1)  M    10   ALIVE    SWORD
Hero(2)  H    10   ALIVE    3    1
```

There are two halves to what is produced.

First, an image of the map from **worldString().**

Second, this is followed by tab (\t) separated table of the entities (Monsters/Heroes) loaded from the file. There are 5 columns in this data.

My header looks like “NAME \tS\tH\tSTATE\tINFO\n".

The first is the type of the entity, then the symbol of the entity, then health, state (ALIVE/DEAD), and finally, information unique to that entity type. For monsters this will be the WeaponType and for heroes this will be their weapon strength, followed by their armor strength.

The data for **gameString()** is accessible from **this.entities**.

public String worldString()

is a **World** method to create the map at the top using the 2D array of entities (**Entity**[][]) stored in the field of the current (**this**) **World** object. The data for **worldString()** is accessible from **this.world**.

The grid will be surrounded by **WALL** (‘#’).

DEAD entities will be indicated by (‘$’).

FLOOR where there is a **null** for an **Entity** ('.').

mvh.enums.Symbol stores these 3 symbols for you as 'constants' in an enumeration type. Symbol.**FLOOR**.getSymbol() will get you the floor symbol '.' for example.

ALIVE entities will be indicated by their symbol as entered by the user in the input file.

The value of this method is so that the Main can advance the game, and for each simulation step Main can output the state of the game. (with this complete, you also now should have an easier time testing/debugging your code as you move forward).

In Reader.java

public static World loadWorld(File fileWorld)

Class-level procedural function (static method) that will open the provided **File** and attempt to create and return a **World** from it. An example input file 'world.txt' was given earlier.

The comma-separated value (csv) world.txt file is as follows:

1. **rows** on the first line
2. **columns** on the second line
3. then it should loop **rows*columns** number of following lines.

   Each following line should start with **row,col**. The first row will be read first across all the columns left to right (0 -> column_count-1). Then the next row, until row_count-1). This means in a 3x3 world we'd read row 0 first and read 0,0 then 0,1 and finally 0,2 before moving on to row 1.

   1. An empty location (null), which is just Floor, is indicated by no data following the **row,col**.
   2. If data follows the **row,col** then the data should be for a **MONSTER**, or **HERO** type.

- Both types expect a symbol to follow the declaration of type. In the example for the **Monster** we chose **M** but this could be any single letter the user wants. After the symbol both types will expect a positive integer health number. In the example, both had a health of 10.
- After that, the remaining entries will be different depending on if the entity is a **HERO** or a **MONSTER**.

1. Monsters have a single option of a single letter for (S)word, (C)lub, or (A)xe **WeaponType** which will determine their weapon strength. Monsters will all have an armor strength of 2.
2. Heroes have two positive integers. First weapon strength and second armor strength.

Monster/Hero input line format:

Monster line

row,col,MONSTER,symbol,health,S/A/C

Hero line

row,col,HERO,symbol,health,attack,armor

You need to read this file. Create a **World** object. This world will store **null** for all the grid locations to start. This is an empty 2D array of floor locations. You will need to store **Hero**/**Monster** objects into this world at the indicated locations with the provided properties replacing these null spots. To create **Hero**/**Monster** objects you will need to use their constructors properly and should look at the **Entity**/**Hero**/**Monster/WeaponType** files provided to understand these classes.

At this point you can load a file and see the World printed at the start of the program.

In Hero.java

public Direction attackWhere(World local)

Heroes receive a 3x3 local world view from the **world.getlocal(3, row, column)** function centred on themselves. From this they need to indicate where they plan to attack. **Heroes** will scan the provided 3x3 local **World** grid (top to bottom) in rows and (left to right) in each row. A **Hero** will attack the first **ALIVE Monster** they encounter. A **Hero** will return null if they do not find a alive **Monster** in one of these locations. They will ignore walls/floor spots, other **Heroes**, and **DEAD** entities. You can use **Direction**.**getDirection** to determine what **Direction** is of attack would attack a row,column adjustment value.

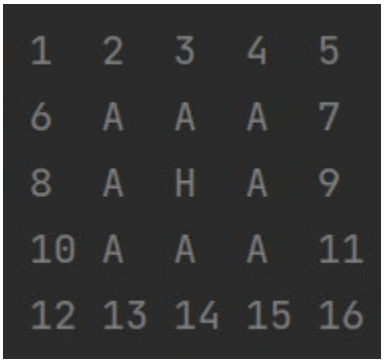
This is an example of the order of scanning 1,2,3,4,5,6,7,8. Location 0,0 is at scanning spot 1 and would have adjustment values of -1 in row, and -1 in column and

Direction.getDirection(-1,-1) would return **NORTHWEST**.

public Direction chooseMove(World local)

Heroes receive a 5x5 local **World** view from the **getlocal(5,row,column)** function centred on themselves. In a correct working game, **Heroes** are first given option to attack, and only if they return **null** (indicating they have no-one to attack) will they then instead be given the chance to move in the direction returned by this function.

From this local **World** the **Hero** needs to indicate where they plan to move. This move is a **Direction** from the enumeration in mvh.enums.**Direction**. **Heroes** will not move if there is an **ALIVE Monster** in any of the 8 positions around them (as they will have already decided to attack this **Monster** in **attackWhere**). If there is no **Monster** around them, then they will scan the local the provided 5x5 local **World** grid (top to bottom) in rows and (left to right) in each row. A **Hero** will move towards the first **ALIVE Monster** they find. This can be expected to be in the outer ring of the grid as the **Hero** would have attacked a closer **ALIVE Monster**. This would be 1 of the 16 spots given.



If there a **Monster** in one of those spots, then the **Hero** will move in the **Direction** of that **Monster**. **Direction.getDirection** would get us the best move, but since this could be blocked **instead you should use Direction.getDirections** which will give you a list of 3 directions to try. You should only return a **Direction** if the **Direction** is a spot that you can **moveOnTopOf**.

If there is no **ALIVE Monster**, then the **Hero** will move **NORTHWEST**. If the **Hero** is unable to move **NORTHWEST**, they will attempt once to move in one random **Direction.getRandomDirection()**. If they can't move in that **one** random **Direction**, then they will **STAY** where they are.

In Monster.java

(In summary this works like Hero moving/attacking except the order of scanning and preferred default direction to move is inverted.)

public Direction attackWhere(World local)

Monsters receive a 3x3 local **World** view from the **world.getlocal(3, row, column)** function centred on themselves. From this they need to indicate where they plan to attack. **Monsters** will scan the local the provided 3x3 local **World** grid (bottom to top) in rows and (right to left) in each row. A **Monster** will attack the first **ALIVE Hero** they encounter. A **Monster** will return null if they do not find an alive **Hero** in one of these locations. They will ignore walls/floor spots, other **Monsters**, and **DEAD** entities. You can use **Direction.getDirection** to determine what **Direction** is of attack would attack a row,column adjustment value.

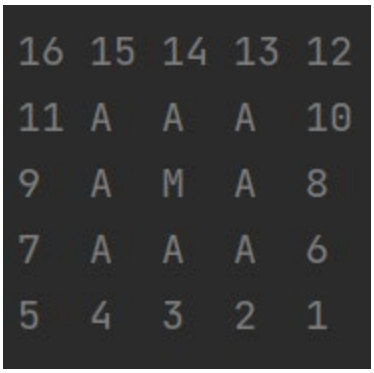
This is an example of the order of scanning 1,2,3,4,5,6,7,8. Location -1,0 is at scanning spot 7 and would have adjustment values of -1 in row, and 0 in column and

Direction.getDirection(-1,0) would return **NORTH**.

public Direction chooseMove(World local)

Monsters receive a 5x5 local **World** view from the **getlocal(5,row,column)** function centred on themselves. In a correct working game, **Monsters** are first given option to attack, and only if they return **null** (indicating they have no-one to attack) will they then instead be given the chance to move in the direction returned by this function.

From this local **World** the **Monster** needs to indicate where they plan to move. This move is a **Direction** from the enumeration in mvh.enums.**Direction**. **Monsters** will not move if there is an **ALIVE Hero** in any of the 8 positions around them (as they will have already decided to attack this **Hero** in **attackWhere**). If there is no **Hero** around them, then they will scan the local the provided 5x5 local **World** grid (bottom to top) in rows and (right to left) in each row. A **Monster** will move towards the first **ALIVE Hero** they find. This can be expected to be in the outer ring of the grid as the **Monster** would have attacked a closer **ALIVE Hero**. This would be 1 of the 16 spots given.



If there a **Hero** in one of those spots, then the **Monster** will move in the **Direction** of that **Hero**. **Direction.getDirection** would get us the best move, but since this could be blocked you should use **Direction.getDirections** which will give you a list of 3 directions to try. You should only return a **Direction** if the **Direction** is a spot that you can **moveOnTopOf**.

If there is no **ALIVE Hero**, then the **Monster** will move **SOUTHEAST**. If the **Monster** is unable to move **SOUTHEAST**, they will attempt once to move in one random **Direction.getRandomDirection()**. If they can't move in that **one** random **Direction**, then they will **STAY** where they are.

Unit Testing Requirement:

Unlike assignment 1, this requirement is much more reduced. You should have 3 tests for the Reader.loadWorld() function, 2 test for world.gameString(), 2 test for world.worldString(), 8 tests total (2 each) for hero.chooseMove(), monster.chooseMove(), hero.attackWhere(), monster.attackWhere().

Bonus

Make a second version of the assignment. This version should expand on the existing one. In particular it should add the ability to designate a class for Heroes and types for Monsters. You should add at least 3 classes for Heroes (Mage, Fighter, Rogue) and 3 types for Monsters (Kobold, Drakes, White Dragon). These classes should result for Heroes in a different weapon/armor strength calculation (weaponStrength()/armorStrength() should work different for each). The Monster types should each have a different armor strength and have their own types WeaponTypes (2 or 3 for each). The Kobolds can easily keep the original 3 types.

Mage/Fighter/Rouge should be sub-class of Hero and Kobold/Drake/White Dragon should be sub-class of Monster. You should be able to indicate who they are in the input world file using a symbol 'M' for Mage, and 'W' for White Dragon for example instead of letting the user use any symbol they want.

At the same time add the ability for the user to add Walls in the middle of the map, as well as traps (which are visible in the regular program map), visible to Monsters, but not visible to Heroes. Monsters should avoid traps by considering them to not be valid places to moveOnTopOf(). Once a Trap is

triggered by a Hero it should cause the damage indicated in the configuration file, and then disappear from the map.

```
3
3
0,0,MONSTER,W,10,C
0,1,MONSTER,D,5,B
0,2,MONSTER,K,2,A
1,0
1,1
1,2,HERO,M,10,3,1
2,0
2,1,HERO,F,10,3,1
2,2,HERO,R,10,3,1
```

The above map has 3 Monsters ((W)hite Dragon, (D)rake, (K)obold) and 3 Heroes ((M)age, (F)ighter,(R)ogue). The White Dragon is using (C)old Breath, the Drake (B)ite, and the Kobold is using (A)xe.

Additional Specification

- You must comment your functions with **Javadoc** comments.
- **Use in-line comments** to indicate blocks of code and describe decisions or complex expressions.
- **Do not use inline conditionals**
- Put your **name, date, and tutorial** into the comments for the **Javadoc** of the class for your TAs to identify your work.
- **Do not rename the provided files.** You must use exactly the requested filenames.
- **You should not import ANY libraries (outside java.io/java.util) to complete the regular assignment. Using these could result in grade of 0 for that portion of the assignment. Citing code from the internet to complete a function will also result in 0.**
- Use constants appropriately. Your TA may note one or two magic numbers as a comment, but more will result in lost marks.
- Do not change the provided code without discussion with the instructor. If a bug or something is broken, the instructor should be informed to fix this issue.
- You should not perform function parameter error checking in this assignment when writing your functions unless specifically requested. The provided code should be designed to assume the rest of the program only provides your functions valid inputs.
-

Grading

The total grade is out of 50. Bonus marks can reach up to 55 marks if completed.
Game Files (out of 25) [grading for TAs will be assisted using instructor unit tests]
world.gameString()/world.worldString() 4
Reader.readWorld() 5
hero.moveWhere() 5

hero.attackWhere() 3
monster.moveWhere() 5
monster.attackWhere() 3
MvHTest.java (out of 15)
Gitlab Usage (out of 5)

Gitlab account exists, private project exists, at least 1 commit, small commits, regular commits

Style/Commenting (out of 5)

Name/Date/Tutorial, Functions commented, Javadoc, Inline commenting, doesn't use inline conditionals, limited magic numbers, don't change function names, don't change filenames, etc.

BONUS MARKS
The type system works (3)
The trap/wall system works (1)
The bonus game works after changes correctly (1)

Invite your TA to your private!!!! csgit.ucalgary.ca project for the assignment

Submit the following using the Assignment 2 Dropbox in D2L

1. Reader/World/Monster/Hero.java
2. MvHTest.java (your unit tests)
3. Name of your private csgit.ucalgary.ca repository
4. (If you did bonus please submit a separate .zip file of your second project clearly labeled)