1.1 Building a Simple Text Combat Game

Tutorial written by Peter Mitchell.

Revision Number 1.0

Contents

1.2 Introduction

This is NOT a graded piece of work! It is just an additional activity that anyone is free to try out if they are interested.

Welcome to this tutorial if you are reading it. Feel free to give feedback. Each section will add some new element to what is being built. Working through this tutorial will demonstrate examples of many fundamentals including arrays, loops, handling user input, polymorphism, and many other useful skills to know as a programmer. After each section of this tutorial the game will gradually have more functionality and be playable.

View a view demo of what the game will look like at: https://youtu.be/7o3ZetXnE3g

Finished version (RPGTextGame folder): https://github.com/Squirrelbear/CP1Extras

Code as it should appear after each step (RPGTextGame_IndividualParts folder in above): https://github.com/Squirrelbear/CP1Extras/tree/main/RPGTextGame_IndividualParts

I would suggest working through and writing the code yourself, but if you get stuck it will show you what the code should look like. Both in the final version, and after each part of writing it.

1.3 High Level Discussion of Requirements

Before delving into writing some code it is a good idea to briefly identify some requirements for what is being built. Writing code blindly without knowing what the goal is can make it necessary to make changes later. There will appear to be a lot of different things in this list of dot points, but this essentially defines how the rules of our game operate. You can skip to the section after this if you would like to start writing code.

- Player
    - Player should have a name that the user can type in at the start of the game.
    - The player's name should be shown when an action occurs caused by that player or when showing the player's status.
    - Player should understand the current and maximum values for a Health stat and Mana stat. The player should also have a way to restore Health and Mana during resting periods.
    - Player should have a selection of multiple attack types they can choose from.
    - Player should have a way to choose which attack type they would like to use, including showing a list of available attacks.
    - The player should be able to take damage when being attacked.
    - The player should be able to spend mana and determine if there is enough mana to perform attacks.
- Enemy
    - Enemy should have a name, health, and a selection of 1 or more attack types.
    - Enemy should have a way to take damage when being attacked.
    - Enemy should have a way to choose an attack from their possible attack types.
    - Enemy should be able to show their status.
    - Enemies should be definable as a particular type of enemy (Orc, Troll, Wolf) with appropriately randomised name, and unique attack types.
- Encounter
    - An encounter is defined as a battle between a player and one or more enemies.
    - When beginning an encounter, a random enemy should be created.
    - An encounter ends when either the player or enemy run out of health.
    - When running an encounter, the player should attack first, and then alternating between the enemy and player equally.
    - The player can only attack if they have enough mana to perform an attack. If they have insufficient mana the enemy can continue attacks until the player is defeated.
    - An encounter should understand how many turns have occurred and show this to the player at the start of each new turn.
    - When either the player or enemy attack a message should be shown with the details about what the attack was, who it was to/from, and how much damage was done.
    - If the player during their stage of the turn defeats the enemy (makes their health 0), the enemy's turn is skipped because they are incapacitated.

- Map
    - A map is defined as a play space consisting of a simple grid of characters representing different elements.
    - An empty cell on the map can be represented as a '.' character.
    - A blocked cell on the map can be represented as a '*' character.
    - An enemy cell on the map can be represented as a '#' character.
    - A player cell on the map can be represented as a '@' character.
    - There can be any number of empty or blocked cells, a small number (or no) enemy cells and only one player cell.
    - The map should print the map as a grid with spaces separating each character.
    - The map should allow input from the user to move the player's @ character.
    - Moving the player's character should only occur if it is a valid move (would not move to a * cell or off the map).
    - When moving to a cell with an enemy (#) an encounter should trigger.
- Game loop
    - The game should start by creating a player, and then a map for the player to play on.
    - The player should be allowed to traverse the map until an encounter is reached.
    - After the encounter is complete, if the player's health reached 0, then the game is over. A message indicating this should be shown.
    - If the player survived the encounter they should be healed, and mana restored by the amounts as defined by the player. Additionally, a counter should keep track of how many enemies have been defeated for each encounter survived.
    - If the map no longer has any enemies to fight, then the game is over, and a victory message should be shown.

1.4 Part One: Getting Started Creating a Player and Game Start

This section will walk through creating your player so that it can be given a name and some stats to represent health/mana. To begin testing this you will also need to create some of the other entry points for the game.

1. Start by creating a new Project in IntelliJ. You can name this whatever you would like, and you should create with the Command Line App template to add the Main java file in. If you don't have the Main.java file in your project after creating it, you should create a new Java file and call it Main.
2. Next create two more Java files. Game.java and Player.java. You can right click on the package under your "src" directory and select New->Java Class then enter each one at a time and press Enter.

3. Open Player.java, we will now begin to add some variables to define what a player is.

4. All the variables should be defined as private (meaning placing the word private before the type). You will need to declare the following variables:
    a. A String type variable called playerName.
    b. An int type variable called playerHealth.
    c. An int type variable called playerMaxHealth.
    d. An int type variable called playerMana.
    e. An int type variable called playerMaxMana.

    The playerName is where we will store the player's name once it has been typed in. The playerHealth and playerMaxHealth represent the current and maximum values for health. The playerMana and playerMaxMana fulfil the same purpose for mana.

5. Create a Constructor for Player. It should take one parameter with the name for the player we are creating. Inside your constructor you should set the value of playerHealth, playerMaxHealth, playerMana, and playerMaxMana all to a value of 100. And then set your playerName in your instance variables to the value passed to your method. This will look something like:

```
public Player(String playerName) {
        // set int variables all to 100
        this.playerName = playerName;
}
```

The "this" keyword allows you to access the variable with the same name in your class and set it to the local variable in your method.

6. Create a toString() method that will return a String type variable defining the current status of our Player class. You should create a String using all the variables we defined in step 4 such that it would look like the following if printed out. You can see the variables are Peter and all the 100 values.
   Player Status: Peter has 100/100 HP and 100/100 Mana.

   The toString() method if you are not sure would look like the following where you would add all the other text and variables to where playerName is.

```
public String toString() {
        return playerName;
}
```

   This will allow us to print out our players status easily later.

7. You can now open the Game.java and set up our Player ready to be used.
   Start by creating two instance variables at the top of your class. Declared as the following (with the private visibility modifier).
   - A Player type variable called player.
   - A Scanner type variable called scan.

   We will be reusing both variables a lot, so it is useful to define them as part of our class.

8. Create a Constructor method for Game. Inside this method you should assign your Scanner. It should appear as seen below.

```
public Game() {
        scan = new Scanner(System.in);
}
```

9. Create a method called createPlayer(). This method should prompt the user with some text and then allow the user to enter their preferred name. You should then read the whole line of text and use the trim() method as seen below to remove any excess whitespace from the start/end. Finally the last step is to create a Player type object and pass it the name we read in.

```
public void createPlayer() {

        // Print a welcome message

        // Print a prompt asking the user to enter a name

        String playerName = scan.nextLint().trim();

        player = new Player(playerName);

}
```

   When you run your write your messages try to make them mimic the output as follows. Where "Peter" is the place your name is typed in.

   Welcome new player!

   Enter your name: **Peter**

10. Create a method called startGame() and call the createPlayer() method inside it.

11. Open your Main.java. If you do not have a main method currently in this file you should add one. Then inside your main method write the following two lines of code:

    Game game = new Game();
    game.startGame();

    This will create a Game object allowing us to call its startGame method. The startGame method can then call createPlayer() to initialise our Player object.

12. To test what we have done is correct we can print out the status of our player. Open the Game.java file again and just after your call to createPlayer() in startGame() you should write the following line:
    System.out.println(player);

    This will call the toString() method in our player object and should print out all the details correctly.

13. You should now compile and run your program. If you have completed all the steps correctly you should see output that looks like the below.

    Welcome new player!

    Enter your name: **Peter**

    Player Status: Peter has 100/100 HP and 100/100 Mana.

1.5 Part Two: Creating our Basic Enemy

Now that we have a player, we should begin creating something for the player to fight against. In this section we will just be creating the start of our base Enemy class that in later steps will be used to define more specific types of enemies.

14. Create a new Java file called Enemy.java.

15. Declare two instance variables (as private visibility):

    - An int type variable called enemyHealth.
    - A String type variable called enemyName.

16. Create a Constructor for the Enemy class with parameters for enemyName and enemyHealth. This should look like to what is seen below.

```
public Enemy(String enemyName, int enemyHealth) {
    // set the instance variables for each variables to those stored in the local variables
}
```

17. Create a toString() method in the like how you did for Player. The toString() method should return a String that looks like:
    Enemy Status: name has health HP.
    Where name and health would be the respective values.

18. To test our Enemy class has been set up correctly we can add some code to the end of the startGame() method where we printed out the player's status. Add the following two lines, although you can choose any values if you wish to make it say something else.

```
Enemy enemy = new Enemy("Bob", 50);
System.out.println(enemy);
```

19. Compile and run your program. You should see similar output to the text below where the only difference from step 13 should be the addition of an enemy status line.

```
Welcome new player!
Enter your name: Peter
Player Status: Peter has 100/100 HP and 100/100 Mana
Enemy Status: Bob has 50 HP.
```

1.6 Part Three: Creating an Attack Type

Now that we have a basic definition for an Enemy and a Player we should start to work toward letting them attack each other. Each attack option for both players and enemies will be defined to have a name, a random range for min/max attack, and a mana cost. The mana cost will only affect the player in this tutorial, but you could make Enemies take it into account if you wish.

20. Create a new Java file called AttackType.java. This file will be used to define the properties we talked about above.

21. Declare the following (private) instance variables as part of your class.

    - A String called attackName.
    - An int called damageValueMin.
    - An int called damageValueMax.
    - An int called manaCost.

22. Create a Constructor for AttackType that sets all four of these variables like you did for the Player/Enemy classes. The method definition should match what is seen below.
    public AttackType(String attackName, int damageValueMin, int damageValueMax, int manaCost) {
    // set each of the instance variables to their respective values
    }

23. We will need to access different parts of this class using a few different methods. The first that you should create is a method called getAttackName(). The method should have a String return type and return the value of attackName.

24. Next create a method called getManaCost(). This method should have an int return type and return the value of manaCost.

25. Create a toString() method like you have done for Player and Enemy. This time you should make the String return with something like the text below. We will be using the toString() to print out the list of available attacks in a later step. The order of values show below are the attackName, then the damageValueMin, then the damageValueMax, and finally the manaCost.
    name (5-10, costs 15)

26. The last method you will need to create for the attack is one to generate a random number in the correct range. You may remember the formula from tutorials for defining a random number within the range of min and max is as follows.
    rand.nextInt(max – min + 1) + min

    Create a method called getDamageValueInRange. It should return an int type and type one Random type parameter called rand. You should use the appropriate variables from your instance variables and the rand variable passed as a parameter to generate an appropriate random number. The method should look like the following.

```
public int getDamageValueInRange(Random rand) {
                    // return your random result here
}
```

27. The AttackType.java file is now complete. You may have noticed we have not yet created our Random object yet. We will set this up now. Open Game.java and you will need to add a private instance variable to the top of your file. The variable should be declared as rand with type Random. Then to initialise the value of our rand object you can do this inside your Constructor. Just after the scan variable has been assigned add in the following line.
    rand = new Random();

28. We should create an example AttackType to test that the methods are working as expected. In Game.java and just after where you printed out the enemy status add the following code. You can change the values to anything else if you would like to try something different.

```
AttackType sampleAttack = new AttackType("Zap", 10, 15, 5);
System.out.println(sampleAttack);
System.out.println(sampleAttack.getAttackName() + " costs "
                    + sampleAttack.getManaCost());
System.out.println("Example damage output: "
                    + sampleAttack.getDamageValueInRange(rand));
```

29. Compile and run your program. If everything is working correctly you should see output like the below. Where it should only change depending on the name you type in and the random number generated in the last line.

```
Welcome new player!
Enter your name: Peter
Player Status: Peter has 100/100 HP and 100/100 Mana.
Enemy Status: Bob has 50 HP.
Zap (10-15, costs 5)
Zap costs 5
Example damage output: 10
```

1.7 Part Four: Adding Attack Types to the Player

In this section we will be adding some AttackType objects to the player and creating a way for the player to choose an attack with input validation.

30. Open Player.java again. We can create an array of 5 different attacks. These can realistically have any name, any damage range, and any mana cost. If you were wanting to make a fun game trying to balance these numbers would be an important aspect. The example attacks recommended here are slightly balanced to give a small chance of losing, with mostly overwhelmingly high chances of winning.

    Create a new instance variable at the top of your Player class just under the other variables you already declared. This variable should be a private AttackType array called attackTypes. To do this you would write the following line.

    private AttackType[] attackType;

31. Now we can define the 5 attack types for the player. The following code is going to be added at the end of the Player Constructor.
    attackTypes = new AttackType[5];

    Then to define the 5 different Attack types they would be written like the following line.
    attackTypes[0] = new AttackType("Magic Missile", 5, 10, 5);

    Use the following table to complete the other 4 Attack types.

Index	attackName	damageValueMin	damageValueMax	manaCost
0	Magic Missile	5	10	5
1	Lightning Bolt	10	20	10
2	Fireball	5	30	15
3	Blinding Flash	10	10	8
4	Tsunami	0	40	20

32. Create a method called printAttackList() that does not return anything and takes no parameters. This method will loop through the array we just populated to print them out as a numbered list. The following loop will step from index 0 to the last index.

    for(int i = 0; i < attackTypes.length; i++) {

    }

    Then to print out each element by using the value of i for the number and use the toString method to print out the information about the attack type. This would look like:

    System.out.println((i+1) + ". " + attackTypes[i]);

33. To test our printAttackList() is working properly open Game.java and add the following line to the end of your startGame() method.
    player.printAttackList();

    Compile and run your game. At the end of your output, you should see the following output:
    1. Magic Missile (5-10, costs 5)
    2. Lightning Bolt (10-20, costs 10)
    3. Fireball (5-30, costs 15)
    4. Blinding Flash (10-10, costs 8)
    5. Tsunami (0-40, costs 20)

34. An important part of making code more readable is splitting it up into sensible divisions with relevant names. To allow the player to choose an attack we will create three methods. The chooseAttack method will be public to allow other methods to call it, and then it will call two helped methods. These provide specific functionality that only is relevant in the context of the class. These are chooseAttackNumber and isAttackValid. The idea is that we will continue attempting to request a valid number that matches a number on the list of attack types presented to the player.

    We will start by defining the chooseAttackNumber method. This should be defined as private with an int return type and a single Scanner parameter named scan. This should look like:
    private int chooseAttackNumber(Scanner scan) {

    }

35. The purpose of this method is to read in a valid integer. This method will not check anything to do with the range of our number.

    Declare and initialise a variable called inputIsNumber and set the value to false. We will use this variable to keep looping until a valid integer has been entered.

    Write a do while loop that has the condition !isInputNumber. This will look like the code below.

    do {

    } while (!inputIsNumber);

36. Inside the loop we need to print out a message indicating what options there are for attacks, ask the player which attack they would like to use, then check the value that has been entered and either display an error or exit the loop.

    Start by calling the method we already defined earlier called printAttackList().

    Next display a message clearly showing the valid range of attacks. You could do this by printing a String like:

    "Which attack would you like to use (1-" + attackTypes.length + "): "

    This makes the method future proof in case we decide to change the number of attackTypes later.

    We do not need to read the value to check what has been entered. The Scanner class has methods like hasNextInt() that can check if the next input is a valid int type. We can write the rest of our loop's content with an if statement checking if the next input is a valid int and either changing our loop variable or displaying an error and skipping that input.

    ```
    if(!scan.hasNextInt()) {
        // print out the message "You can only enter numbers in the valid range for this input!"
        scan.nextLine();    //  this will throw away the invalid input on that line.
    } else {
        inputIsNumber = true;
    }
    ```

37. The last step for creating this method is adding a return statement to the end. This will be reached when the loop exits which only happens when a valid int type has been typed in. This means we can write the following line to read that int value and return it.

    ```
    return scan.nextInt();
    ```

38. The second private helper method we will need is a method that can check if the number collected from the previous method is a number in the valid range, and the player has enough mana to cast it. Your method should be declared as seen below.

    ```
    private boolean isAttackValid(int choice) {
        // remaining code will be written here
        return true;
    }
    ```

39. The order of validation is important. To check how much mana it will cost we need to validate that the player's choice is within the range of array elements. Write an if statement that checks if the choice is less than 1 or greater than the length of the attackTypes array. If one of these conditions is true print out the following error message text.
"Invalid choice. Please enter a number in the valid range."

    After printing the message you should return false to indicate that this choice was not valid.

    To prevent the player from using attacks that cost more mana than we have the cost of the chosen spell can be compared to the playerMana. To access the mana cost of the chosen attack you can write the following code.
    attackTypes[choice-1].getManaCost()
    Notice that we subtract one from the choice to shift from the 1 to 5 that is shown to the player to the 0 to 4 representing indexes in the array.
    Write another if statement after your previous if statement that checks if the mana cost of the chosen spell is greater than the playerMana. If this is true the following error text can be printed followed by another return false.
    "You do not have enough mana to cast that attack. Select something else."

40. Now that the two methods have been created as private helper methods, the public chooseAttack method can be written. The following code shows what you should write for this method.

```
public AttackType chooseAttack(Scanner scan) {
        int choice;
        do {
                choice = chooseAttackNumber(scan);
                scan.nextLine();
        } while(!isAttackValid(choice));
        return attackTypes[choice-1];
}
```

    Briefly looking through what this method is doing, you can see it has another do while loop that is used to read in our attack number from the first method we defined. It then sanitises our input with a scan.nextLine() call for whenever we next have to input something. The loop ends when the chosen attack is determined to be valid by the other method we wrote. The final step returns the chosen attack from the attackTypes array.

41. You have added a lot of code over the past few steps so let us test that the chooseAttack method works. Open your Gama.java file. Before testing the chooseAttack method we should clear up the growing amount of test output. You should either delete or comment out lines of code in your startGame() method until It shows as:

```
public void startGame() {

        createPlayer();

}
```

After the createPlayer() line add in the following lines of code:

```
while(true) {

        AttackType attackType = player.chooseAttack(scan);

        System.out.println("You selected the " + attackType.getAttackName() + " attack!");

}
```

This will let you infinitely loop and test a variety of inputs to make sure it performs as expected. You should try some of the following example inputs to make sure the output appears as you would expect.

- Input: "a"
  Output: "You can only enter numbers in the valid range for this input!" followed by the question being re-asked.
- Input: "-1"
  Output: "Invalid choice. Please enter a number in the valid range." Followed by the question being re-asked.
- Input: "6"
  Output: same as above.
- Input: "1"
  Output: "You selected the Magic Missile attack!"

It is currently not possible to test the mana cost, but you could change the value stored in playerMana to be lower than at least one of the options if you wish to test it at this time.

1.8 Part Five: Creating Enemy Variations (Orc, Troll, Wolf)

The Enemy class provided a basic first implementation of our Enemy base class. Before creating variations of enemies, the Enemy base class needs to be improved to include the new AttackType information. Once the additional changes have been made we will be creating three additional classes to support more specific types of enemies by extending the Enemy base class.

42. Open Enemy.java and add the following:
    - A new private instance variable of type AttackType[] called attackTypes. This will appear identically as what you should have written in the Player class.
    - Add another parameter to your Enemy constructor of type AttackType[] called attackTypes.
    - Add a line to the end of your constructor to assign the value of attackTypes to the instance variable by writing: this.attackTypes = attackTypes.

43. While we still have the Enemy class open there are two additional methods that would be useful to include to be used in future steps. These are a chooseAttack() method and a getEnemyHealth() method. Define the methods as shown below.

```
public AttackType chooseAttack(Random rand) {
        return attackTypes[rand.nextInt(attackTypes.length)];
}
public int getEnemyHealth() {
        return enemyHealth;
}
```

    The chooseAttack method is going to choose a random attack from those available to the enemy. It does not care what the mana cost is for those attacks or provide any strategy around how attacks should be selected.

44. Create a new Java file called Orc.java. This class will take the base Enemy class and provide some unique AttackTypes and a suitable random name.
    Change the line that defines the class to make it say the following. By extending the Enemy this class will gain all the functionality of the Enemy class.
    public class Orc extends Enemy

45. The Orc will contain no additional instance variables and there will be three methods. Start by defining a constructor for Orc with the definition below.

```
public Orc(Random rand) {
        super(createRandomOrcName(rand), 100, createOrcAttacks());
}
```

The call to super() will call the Enemy constructor that requires a name, health, and list of attacks. The two methods you can see in this super call are what will create our unique name and attack list for suitable for an Orc.

46. The first method that was called in the constructor above is the createRandomOrcName(). Declare a method called createRandomOrcName that takes a parameter of type Random called rand and returns a String. You will also need to make the method private and static. This will look like the code below.

```
private static String createRandomOrcName(Random rand) {

}
```

Declare a String called name and initialise it to "(Orc)".

Create a switch statement that switches based on rand.nextInt(3). You will need to write three cases for each of 0, 1, and 2. Inside these cases you will need to assign the value of name line the code below.

```
name = "examplename " + name;
```

This will put the (Orc) after the name. Make sure you put a break; after each of these. You can choose to use any names you would like, but the table below are the names used in developing this tutorial.

Case	Name
0	"Draka "
1	"Thrall "
2	"Garrosh "

The last step to complete the method is to return the name variable.

47. Declare a method called createOrcAttacks() that takes no parameters, returns an array of type AttackType, and has private and static modifiers. This should appear like below.
    private static AttackType[] createOrcAttacks() {
    }
    Declare an array of type AttackType called attackTypes and initialise it to an array with 3 elements.
    AttackType[] attackTypes = new AttackType[3];

    Declaring your three array elements will be the same as how you created them for the Player. You can refer to your Player code if you need a reminder. You can follow the table below for values to create these AttackType objects with.

Index	attackName	damageValueMin	damageValueMax	manaCost
0	Swing	3	5	0
1	Thwack	5	9	0
2	Roar	1	4	0

    You will notice these values are much lower than those used for the player to make it easier to survive. The manaCost is set to 0 because it will not be used for the Enemy objects, but it could be used later to provide a more complex Enemy.

    The last step after assigning the three attackTypes is to return attackTypes from the method.

48. Create a new Java file called Troll.java. The Troll class will be functionally almost identical to the Orc with different attacks and different names.
    Add the “extends Enemy” after your Troll class name as you did for the Orc.

49. Open your Orc.java and copy the three methods. Paste these into the Troll class. And then follow the instructions below to adjust this content.

    - Rename the Orc constructor to Troll.
    - Rename the two method calls inside the constructor to createRandomTrollName() and createTrollAttacks().
    - Rename the method declarations of the two methods to match the method calls above.
    - Modify the createRandomTrollName method to initialise the name String to "(Troll)".
    - Modify the three names based on the table below.

Case	Name
0	"Zul'jin "
1	"Jin'zakk "
2	"Vol'jin "

    - Modify the three AttackTypes based on the table below (Only the names change).

Index	attackName	damageValueMin	damageValueMax	manaCost
0	Spear Throw	3	5	0
1	Voodoo	5	9	0
2	Stealthy Slash	1	4	0

50. Create a new Java file called Wolf.java. The Wolf class is the last type of Enemy we will be defining. Add the "extends Enemy" as you did in the previous two class definitions.

51. Follow the same steps as before with copying from either the Orc.java or Troll.java. Then replace all the Orc or Troll words with Wolf. See step 49 for a summary of all the places you need to change.

52. Modify the names to match the table below.

Case	Name
0	"Shadow "
1	"Goremaw "
2	"Ironjaw "

    Modify the Attack types based on the table below (Only the names change).

Index	attackName	damageValueMin	damageValueMax	manaCost
0	Bite	3	5	0
1	Vicious Bite	5	9	0
2	Howl	1	4	0

53. Now that we have three types of Enemy defined, we should test that they output information as expected. We will create a spawnRandomEnemy() method and use this to quickly see a variety of different enemies.
Open Game.java and create the following method.

```
private Enemy spawnRandomEnemy() {
        Enemy enemy = null;
        switch(rand.nextInt(3)) {
                case 0:
                        enemy = new Orc(rand);
                        break;
                case 1:
                        enemy = new Troll(rand);
                        break;
                case 2:
                        enemy = new Wolf(rand);
                        break;
        }
        return enemy;
}
```

This method will create a single random enemy based on the three classes we defined.
In the startGame() method commend out or delete the while(true) loop that was added at the end of last part. You should only leave the createPlayer() method call line.
Then add the following code. It will create 20 different random enemies and print out the enemy status for each.

```
for(int i = 0; i < 20; i++) {
        Enemy tempEnemy = spawnRandomEnemy();
        System.out.println(tempEnemy);
}
```

54. Compile and run your program. You should see a list of 20 random enemies where you should see a variety of different names that include most of the 9 different names we created over this part.

1.9 Part Six: Creating Encounters with an Enemy and Player

An Encounter for the purpose of this game is going to be a battle between a single Player and a single Enemy that ends when either combatant runs out of health. This part will mostly be preparation for part seven when the actual combat will be added.

55. Create a new Java file called Encounter.java. Start by declaring the following four private instance variables.
    - A variable of type Player called player.
    - A variable of type Enemy called enemy.
    - A variable of type Random called rand.
    - A variable of type Scanner called scan.

56. Create a constructor for Encounter that takes three parameters. A variable of type Player called player, a variable of type Random called rand, and a variable of type Scanner called scan. This should appear as follows.
    ```
    public Encounter(Player player, Random rand, Scanner scan) {
    }
    ```

    Inside the constructor assign the three parameter values to the appropriate instance variables. And then write the following line.
    ```
    enemy = spawnRandomEnemy();
    ```

57. Open Game.java and cut the spawnRandomEnemy() method we previously created. Paste it at the end of your Encounter class. We will be using this method to create enemies when an encounter is created as was added into the constructor during the previous step.

58. Open Player.java and create a public method called getPlayerHealth() that takes no parameters and returns an int. The method should return the value in the playerHealth instance variable. This will be needed for some of the logic to be added for the Encounter class.

59. Open Encounter.java and declare a method called performPlayerTurn() that takes no parameters and returns nothing. For now, just add the following line to the method.
    ```
    AttackType attackType = player.chooseAttack(scan);
    ```
60. Declare a method called performEnemyTurn() that takes one parameter of type Enemy called enemy and returns nothing. Add the following line to the method.
    ```
    AttackType attackType = enemy.chooseAttack(rand);
    ```

61. Declare a method called performAttack() that takes one parameter of type AttackType and returns nothing. This method will be changed later to deal the damage, but for now it can be used to output an amount of damage and an attack name.

    Inside performAttack() declare a new variable of type int called actualDamage and assign it the result of calling attackType.getDamageValueInRange(rand).

    Write code to print out a message using attackType.getAttackName() and actualDamage to print a line like the following example output.

    Used attackName for actualDamage damage.

    Used Magic Missile for 8 damage.

62. Add the following line to the end of your performPlayerTurn() and performEnemyTurn() methods.
    performAttack(attackType);

63. Declare a method called runEncounter() that takes no parameters and returns nothing. This method will be sequencing turns between each player until a survivor is decided.

    Start by printing out a message that says: “Starting new encounter”.

    Declare a variable called turnNumber with type int and initialise its value to 1.

    Declare a while loop that checks if the player’s health is greater than 0 and the enemy’s health is greater than 0. You should use the getPlayerHealth() method from player and getEnemyHealth from enemy to write this condition.

    The remaining code will be inside the while loop.

    Print a line of text with the following message where turnNumber would be using the appropriate variable.

    Turn turnNumber beginning!

    Print out the enemy and player status using the following line.

    System.out.println(enemy + “\n” + player + “\n”);

    Call the method performPlayerTurn().

    Write an if statement to check if the enemy health is less than or equal to 0. The purpose of this statement is to skip the enemy’s turn if they have already been defeated during this turn number. If this condition is true print out the following message.

    You have defeated the enemy!

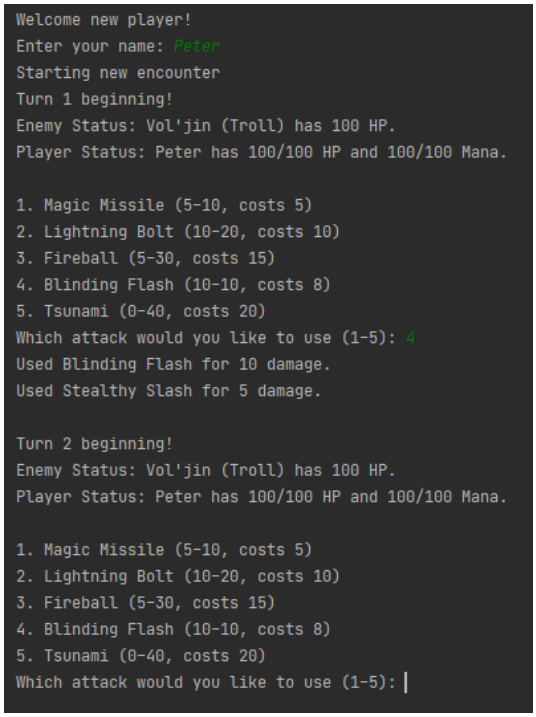
    Add an else statement after the println. Inside the else you should call performEnemyTurn passing enemy as a parameter.

    After the else you should add a println method call with no parameter such that an empty line is printed. And lastly increment turnNumber by 1.

64. Open Game.java so we can add some code to test creating an Encounter. Comment out or delete code inside your startGame() method so that it appears as seen below.

```
public void startGame() {
        createPlayer();
        Encounter encounter = new Encounter(player, rand, scan);
        encounter.runEncounter();
}
```

Compile and run your program. The following is an example of what you should expect to see as output. This will currently run infinitely because the health will not change yet. Make sure that you see different attacks being used and see damage values appearing as expected.



1.10 Part Seven: Dealing Damage by Making AttackableTargets

To make Enemy and Player objects understand how to deal damage to each other we will use an interface. This will allow us to apply some useful polymorphism to reduce the code.

65. Create a new Java file called AttackableTarget.java. When you are naming the class, you can select “Interface”. If you did not click Interface you should change the word “class” to “interface” at the top of your file.

66. Declare the following two lines inside the AttackableTarget interface.

    ```
    void takeDamage(int amount);
    String getName();
    ```

    These methods will allow a defined way for an enemy to attack a player and vice versa. While also providing a way to get the name of the target. This could be made more complex with more methods, but just these two will be suitable for this example project.

67. Open the Player.java file. At the very top change the class definition line so it is written as the following.
    ```
    public class Player implements AttackableTarget
    ```

    This will require the Player class to implement both the methods we just defined in the AttackableTarget interface. You can either press Alt+Enter and select Implement Methods to auto fill the missing methods. Or you can create the methods as follows.
    ```
    public void takeDamage(int amount) {

    }

    public String getName() {

    }
    ```

    The takeDamage method should subtract the variable amount from playerHealth.
    ```
    playerHealth -= amount;
    ```

    The getName() method should return the playerName instance variable.

68. Open the Enemy.java and complete the same steps as you did for Player. Where you would implement AttackableTarget, modify the value of enemyHealth by amount in takeDamage(), and return the value of enemyName in getName().

69. Open Encounter.java. Now that there are accessor methods for the names modify the println statements in runEncounter to be more informative.

    - Modify “Starting new encounter” to say “Starting new encounter against “ + enemy.getName().
    - Modify “You have defeated the enemy!” to say “You have defeated “ + enemy.getName() + “!”.

70. Modify the definition of performAttack to have the following parameter list.
    public void performAttack(AttackableTarget attacker, AttackableTarget defender, AttackType attackType)

    By defining the attacker and defender as the generic AttackableTarget we can pass the player or enemy to these variables in any order, and then give relevant combat messages using the getName() method along with calling the takeDamage method to apply the actualDamage.
    Modify the println statement you wrote in the performAttack to print the following message.
    attacker.getName() + “ attacks “ + defender.getName() + “ with “ + attackType.getAttackName() + “ for “ + actualDamage + “ damage.”
    This will print out messages like:
    Peter attacks Vol’jin (Troll) with Magic Missile for 8 damage.
    Or if the enemy was the attacker it may look like:
    Vol’jin (Troll) attacks Peter with Stealthy Slash for 3 damage.

    To make the defender take the damage you can add the following line to the end of your performAttack method.
    defender.takeDamage(actualDamage);

71. The performAttack method was modified in the previous step, so it is now necessary to change the two places where we called it previously in performPlayerTurn() and performEnemyTurn().

    Change the call to performAttack in performPlayerTurn() to the following.

    performAttack(player, enemy, attackType);

    Change the call to performAttack in performEnemyTurn() to the following.

    performAttack(enemy, player, attackType);

72. Compile and run your program. Test to make sure you get a random enemy, test each of your different attacks, verify the output appears as expected, including showing the outcome of each attack from you and the enemy, make sure the health of your player and the enemy are decreasing, and finally that the victory message saying you defeated the enemy once reaching the end.

1.11 Part Eight: Creating the Game Loop and Using Mana

So far, the parts we have been writing have gradually added individual single elements without a way to continue between multiple encounters. Eventually the player will be moving between Encounters using a map. For now, this part will be focused on creating a loop that starts encounters and then manages the game state between these encounters. Currently the player does not use mana to perform their attacks and the player does not recover any health/mana between encounters. Those will be improved during this part.

73. Open Player.java and start by adding three additional private instance variables to the top of your class.
    - A variable of type int called minimumSpellCost.
    - A constant variable of type double called HEAL_HP_MULTIPLIER, initialise it to 0.3.
    - A constant variable of type double called MANA_RESTORE_MULTIPLIER, initialise it to 0.8.

    The minimumSpellCost variable will be used to check if at least one spell can be cast by the player based on how much mana they currently have. The HEAL_HP_MULTIPLIER and MANA_RESTORE_MULTIPLIER will be used to provide 30% health and 80% mana restoration after an encounter.

74. Declare a method called spendMana that takes one parameter of type int called amount that returns nothing. This should just subtract amount from the playerMana in the same way it was done for the takeDamage method.

75. Declare a method called healPlayer() that takes no parameters and returns nothing.

    Check if the playerHealth is equal to playerMaxHealth. This means it is not necessary to heal the player. Print the following message and then call return; to terminate the method.
    "Heal was cast, but your health was already full."

    Calculate the amount to heal using the following line.
    int healAmount = (int)(playerMaxHealth * HEAL_HP_MULTIPLIER);

    Healing for this amount could give more than the maximum health. To prevent this, it is necessary to cap the amount healed based on the playerMaxHealth. Write an if statement to check if the playerHealth + healAmount is greater than playerMaxHealth.
    If this condition is true then use the following line to set healAmount to the max value.
    healAmount = playerMaxHealth – playerHealth;

    Add the healAmount to the player's health by writing.
    playerHealth += healAmount;

    Write code to print the following message to the player replacing appropriate text with variables.
    You have been healed for healAmount bringing your health to playerHealth/playerMaxHealth.

    This should appear for example as output like.
    You have been healed for 30 bringing your health to 100/100.

76. Duplicate the healPlayer() method. Rename the duplicated method to restoreMana(). The functionality will be identical, but of course the variables and messages will all need to change. The following list summaries the changes you should make.
    - Change all places in the method where it says playerHealth to playerMana.
    - Change all places in the method where it says playerMaxHealth to playerMaxMana.
    - Change HEAL_HP_MULTIPLIER to MANA_RESTORE_MULTPLIER.
    - Change the first message to:
      "Restore Mana was cast, but your Mana was already full."
    - Change the second message to:
      You have been restored for healAmount Mana bringing your Mana to playerMana/playerMaxMana.

    Make sure you look carefully to change all the variable names or it may cause logical errors.

77. Declare a method called canAttack() that takes no parameters and returns a boolean. Inside this method return the result of playerMana > minimumSpellCost. This method will indicate if the player can attack based on their current mana.

78. Declare a method called updateMinimumSpellCost() that takes no parameters and returns nothing. Add the following lines of code to the method.

```
if(attackTypes.length == 0) {
        minimumSpellCost = playerMaxMana+1;
        return;
}
minimumSpellCost = attackTypes[0].getManaCost();
for(int i = 1; i < attackTypes.length; i++) {
        if(attackTypes[i].getManaCost() < minimumSpellCost) {
                minimumSpellCost = attackTypes[i].getManaCost();
        }
}
```

    This will first check if there are no attacks added to the array. To ensure this means the player could not attack the minimumSpellCost is set to 1 more than the player's max mana. The remaining code finds the minimum mana cost from those in the attackTypes array. For this example, it should contain the number 5 at the end.

79. The last step for modifying the Player class is to call our updateMinimumSpellCost(). At the end of the Player constructor add a call to the updateMinimumSpellCost() method.

80. Open Encounter.java. The performPlayerTurn() method can be modified to now take the player's mana into account. Add an if statement that checks if the player can not attack using the condition !player.canAttack(). If this condition is true you should print out the following message and then return; to skip the player's turn.

    "You have insufficient Mana to attack. Turn skipped."

    Between the other two lines (after the chooseAttack() call and before the performAttack() call) you should add in the following line to spend the mana based on the selected attack.

    player.spendMana(attackType.getManaCost());

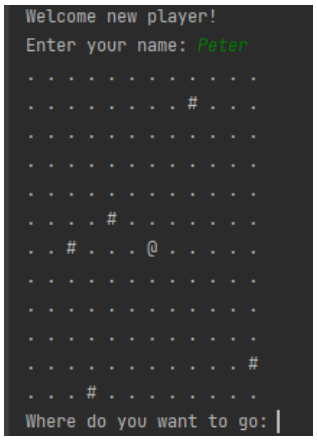
81. Open Game.java. Modify the startGame() method with the following steps.

    - If you have any commented code this would be a good time to move it somewhere else or delete it. You will only need the createPlayer() method call and the two lines we added previously to create an Encounter and run it.
    - After the createPlayer() method call create a new variable of type int called enemiesDefeated and initialise it to 0. This variable will be used to track how many enemies have been defeated so far.
    - Write a while loop that checks the condition player.getPlayerHealth() greater than 0. This loop will end when the player has been defeated.
    - Move the two Encounter lines inside the while loop. If you deleted them, they should be the following two lines.
      Encounter encounter = new Encounter(player, rand, scan);
      encounter.runEncounter();
    - After these two lines inside the loop write an if statement that checks if player.getPlayerHealth() is greater than 0. If this is true you should increment enemiesDefeated by 1, call player.healPlayer() and call player.restoreMana(). This will update the number defeated and restore the player ready for the next encounter.
    - After the end of the loop write the following two messages. Replacing the enemiesDefeated with the variable.
      Oh no! You were defeated.
      You defeated enemiesDefeated enemies before you fell!

82. Compile and run your program. Test the newly added code. You should check that your mana is decreasing each time you attack. When you defeat enemies, a new enemy encounter should start. Try spamming option 5 for Tsunami until you run out of mana to verify that you lose and see several enemies defeated before you lost.

1.12 Part 9: Creating a Map with Encounters

The map we will be creating in this part is a simple ASCII map where a "." represents an empty cell, the "@" represents the player's current position, a "*" represents an obstacle, and a "#" represents an enemy encounter. Note that for the purpose of this task there will not be any "*" obstacles placed, but the mechanics for them will exist. Traversing this map to move the player will be done by typing in a direction (up, down, left, or right). If the player can move in that direction, they will be moved. The map will appear like the image below.



83. Create a new Java file called Map.java. Start by creating the following four private instance variables.
    - A variable of type int called playerX.
    - A variable of type int called player.
    - A variable of type char[][] called map.
    - A variable of int called enemyCount.

    The playerX and playerY will represent the player's current position on the map. The map variable represents a 2D array of characters to store the ASCII version of our map. And enemyCount will be used to keep track of how many enemies remain on the map.

84. Create a constructor for Map that takes one parameter of type Random called rand. For now, inside this method just assign the value 0 to enemyCount.

85. Create a private method called fillNewMap() that returns nothing with two parameters of type int called height and width. Assign map a new array using height and width as the size as shown below.
map = new char[height][width];

Write a for loop using the variable y that starts at y = 0, has a condition of y < map.length and increments with y++.
Inside this for loop add another for loop using the variable x that starts at x = 0, has a condition of x < map[0].length and increments with x++.

Inside of the inner loop you need to set the value at y, x to a '.' using the following line.
map[y][x] = '.';

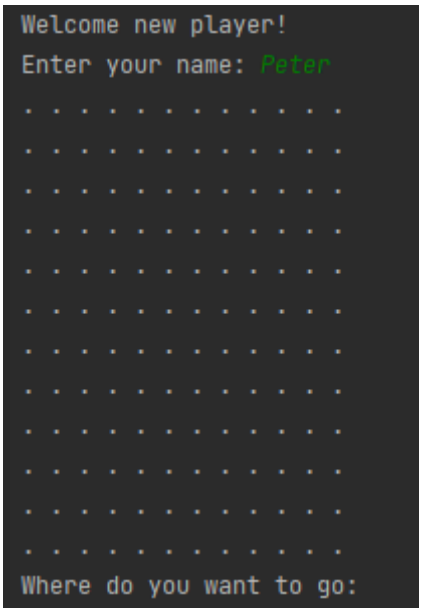
86. Create a method called printMap() that takes no parameters and returns nothing. Copy the loop you wrote in the previous step. Replace the line that sets the cell to a '.' with a print as follows.
System.out.print(map[y][x] + " ");

After the inner loop you should add a System.out.println() by itself. This will add a new line after each row of characters.

87. Open Game.java so we can add code to begin testing the Map is working. Follow the steps listed below to begin testing the map.
    - Add a private instance variable of type Map called map.
    - Just after createPlayer() has been called in startGame() add the lines:
      map = new Map(rand);
      String input;
    - At the start of your while loop add an inner while loop with the condition player.getPlayerHealth() greater than 0. This condition will be changed soon once we have a more sensible condition to use.
    - Inside this loop write the following three lines.
        - map.printMap();
        - System.out.print("Where do you want to go: ");
        - input = scan.nextLine();

88. Open Map.java and add the following line to the start of your constructor.
fillNewMap(12, 12);

89. Compile and run your program. You should see a 12x12 gride of '.' characters with spaces as seen in the picture below.



90. Open Map.java and create a new method called setPlayerPosition it should not return anything and will take two parameters of type int called x and y.
    Set the value of playerX to the value of x.
    Set the value of playerY to the value of y.
    Set the value of map[y][x] to the value '@'.

91. Add the following line to your Map constructor just after the fillNewMap() method call.
    setPlayerPosition(6, 6);

92. To move the player, we will need to define a few different methods. getTranslationFromInput() will take the input from the user and get the relative change to x and y coordinates. canMoveTo() will check that the place being moved to is still inside the bounds of the map and the cell is not an obstacle. getTargetCellEventNumber() will tell us if the cell being moved to is an encounter. swapPlayerToPosition() will perform the swap from the player's current location to the new location. And finally, the movePlayer() method will combine all these methods together to process all the other methods.

    Start by declaring a private method called getTranslationFromInput() that takes one String type parameter called input and returns an int[] type. Write the following line to force any input to lowercase and remove any extra whitespace.

    input = input.toLowerCase().trim();

Write if and else if statements with conditions like the following example using the values shown in the table below.

```
if(input.equals("up")) {
        return new int[] { -1, 0 };
}
```

Input	Output
up	new int[] { -1, 0 }
down	new int[] { 1, 0 }
left	new int[] { 0, -1 }
right	new int[] { 0, 1 }

After writing if and else if statements for all these cases finish with an else that returns new int[] { 0, 0 }.

These numbers represent the change in x or y value to match where the player would need to move to perform an up action for example.

93. Declare a private method called canMoveTo() with two parameters of type int called x and y, and a return type of boolean.
    Write an if statement to check if x is less than 0, or y is less than 0, or x is greater than or equal to map[0].length or y is greater than or equal to map.length.
    If this is true, return false.
    Write another if statement that checks if map[y][x] is equal to '*' representing an obstacle. If this is true, return false.
    Finally return true at the end of the method.

94. Declare a private method called getTargetCellEventNumber() with two parameters of type int called x and y, and a return type of int.
    Write an if statement to check if map[y][x] is equal to '#' then return 1. Otherwise return 0.

    This method would allow you to later add other types of events by associating a character with a specific number.

95. Declare a private method called swapPlayerToPosition() with two parameters of type int called x and y, returns nothing.
    Write an if statement to check if map[y][x] is equal to '#' and if this is the case decrease enemyCount by 1. This is because we are overwriting the cell with the player during this method.
    After the if statement assign the value '.' to map[playerY][playerX].
    Then call setPlayerPosition(x, y);

96. Declare a method called movePlayer() with one parameter of type String called input and that returns an int type. Add the following lines of code that will get the direction of movement and verifying that a movement will happen.

```
int[] translation = getTranslationFromInput(input);
if(translation[0] == 0 && translation[1] == 0) {
        System.out.println("Invalid input. Enter up, down, left, or right.");
        return -1;
}
```

This would show an error if the input were not valid and exit with an error code of -1.

Next, we need to calculate the new location to move and verify we can move there with the following lines.

```
int newX = playerX + translation[1];

int newY = playerY + translation[0];

if(!canMoveTo(newX, newY)) {

        System.out.println("You can't move there. Try move somewhere else.");

        return -1;

}
```

Again, this code will display an error and return from the method with an error code of -1 if the new location is not valid for any reason. After this we can finalise the player moving using the last three lines below.

```
int resultCode = getTargetCellEventNumber(newX, newY);

swapPlayerToPosition(newX, newY);

return resultCode;
```

Before moving the player to the target cell, we get the event number to check if it should trigger an encounter. Then the player's '@' symbol is moved to the new cell and we return the result to indicate if there should be an encounter triggered.

97. Before adding any encounters in we can test moving the player. Open Game.java and modify the startGame() method with the following changes.
    - Right before the while loop for the map that we defined during the earlier steps add declare and initialise a variable called encounterEvent of type int with the value 0.
    - Change the while loop's condition to encounterEvent <= 0
      This is changing the loop to keep going until an event is triggered.
    - After the input = scan.nextLine(); line add the following line.
      encounterEvent = map.movePlayer(input);

98. Compile and run your program. You should be able to move the player around the grid by typing in up/down/left/right. You should test that invalid inputs show the error message correctly. You should also test that you get a message telling you that you cannot move off the grid.

99. To finish off the map we still need to spawn '#' characters to trigger encounters by exiting the loop were just created and create the end condition to be shown when the player defeats all the encounters.

    Open Map.java, and declare a method called allEnemiesDefeated() that takes no parameters and returns a boolean. The result should be returning true if enemyCount is equal to 0.

100. Declare a private method called spawnRandomEnemy() that takes one parameter of type Random called rand and does not return anything.

    Declare an int variable called x and initialise it to a random number using map[0].length.

    Declare an int variable called y and initialise it to a random number using map.length.

    Write an if statement to check if map[y][x] is equal to '.'. If this is true set the value of map[y][x] to '#' and then increment enemyCount by 1.

    This method is generating a random position on the grid and then only placing an encounter at the location if the cell is empty. This does mean the method will not always spawn an enemy.

101. In the Map constructor at the end of the method add the following lines of code to generate up to 5 randomly placed encounters on the map.

```
for(int i = 0; i < 5; i++) {
        spawnRandomEnemy(rand);
}
```

102. Open Game.java and add the following if statement to the end of your if statement near the end of startGame() just after the player.restoreMana() call.

```
if(map.allEnemiesDefeated()) {
        System.out.println("You defeated all the enemies! You win!");
        return;
}
```

103. Compile and run the program. Test that everything works. Make sure that you can traverse the map and entering a '#' triggers the encounters. You should visually see up to 5 '#' characters on the map. Try out clearing all of them to verify the game will end when all enemies have been defeated.

104. As a final useful tool, we will add the ability to type in “quit” at any time as input to quit the program. Open Game.java and declare a public static method called getStringOrQuit() that takes one Scanner parameter called scan and returns a String. Write the following code for the method.

```
public static String getStringOrQuit(Scanner scan) {
    String input = scan.nextLine();
    if(input.equalsIgnoreCase(“quit”)) {
        System.out.println(“Thanks for playing! Goodbye!”);
        System.exit(0);
    }
    return input;
}
```

105. We now just need to find all the places where scan.nextLine() was called and replace it with the method that was just created. This list should include all of them.
   - In Game.java in createPlayer() change the appropriate line to:
     String playerName = getStringOrQuit(scan).trim();
   - In Game.java in startGame() change the appropriate line to:
     input = getStringOrQuit(scan);
   - In Player.java in chooseAttackNumber() change the nextLine() call to:
     Game.getStringOrQuit(scan);
     Move this line before the println so that the error message does not appear if you type quit.

106. Compile and run your program. Verify that you can type quit at the player’s name choice, at the map movement input, and at the number input for attack choice. Verify everything else is appearing to be correct as you do. If you want to test the game properly go back through the high-level requirements defined at the beginning and verify that all have been accounted for.

This concludes the tutorial. I hope you enjoyed the walkthrough and understand what was shown and why each part we included was important to make the game. If you are interested in taking this further the last section will talk about some of the ways you could take this game further.

1.13 Extension: Paths for Improving the Game

You could continue to develop this game and add as many or as few features as you like. The following list elaborates on what you could add to improve the game. It will take a very long time to implement lots of these, but they may provide inspiration.

1) Add obstacles into the map: The map already supports '*' representing an obstacle in the code, but there is no way included to add these. Make sure if you automate this in some way that the player can still reach every encounter on the map.
2) Add additional types of encounters to the map: You could modify the code to have encounters that are not random. You could have a separate character to represent each different type of enemy (e.g., W for Wolf, O for Orc, T for Troll). You would need to modify the Encounter class and Game class to implement this as well.
3) Add additional types of enemies: We only created three types of enemies; you could create any other enemies you would like. Including adding more different names.
4) Perform some game balancing: The enemies all have the same basic attacks, and the player's attacks are not all that different either. You could try to balance the numbers to create a more challenging game.
5) Make attack types do more than just deal damage: You could make attacks that act as a self-heal, hit multiple enemies, apply some sort of buff or debuff, or use a potion from an inventory. This is a complicated extension depending on what you add.
6) Make encounters that support multiple enemies: The encounters we created only have 1v1 battles. There is no reason why the player and/or enemies could not have multiples in their party.
7) Add an inventory with items: You could place items on the map and collect them as events, then use those to heal between (or perhaps during) encounters.
8) Add player leveling: Each time you defeat an enemy you could gain some amount of experience and then after an amount of experience is reached you would level up. This could increase your health, mana, and modify your attacks.
9) Add levels to enemies: Like the player leveling you could make appropriate enemies spawn based on the level of the player to provide appropriate challenge.
10) Add modifiers to damage: You could add damage types and armour types, so some attacks are better against some armour types.
11) Add multiple maps: so that when you finish a map you move to the next level. You could add file loading to store your maps or generate them randomly each time.
12) Turn the game into a GUI application: If you want to take it all the way, you could go beyond a text-based game and make it game with a proper interface.