Homework 3: Expanded Combat System

Summary:

This project expands on the previous assignment to build a simple text-based action game. This project focuses on methodizing the processes of your software. There are three tasks that your software must perform.

1. Create a hero
2. Create a monster
3. Run the combat algorithm: hero versus monster

The above defines the basic fundamental structure for a complete game. In the first phase, the player will decide how to best build their own hero. In the second phase, the system will randomly generate a monster such as a goblin, orc, or troll. In the final phase, the system will execute a turn-based combat system similar to the one in the previous assignment. You should identify the various responsibilities that govern your software system and model them using methods.

Objectives:

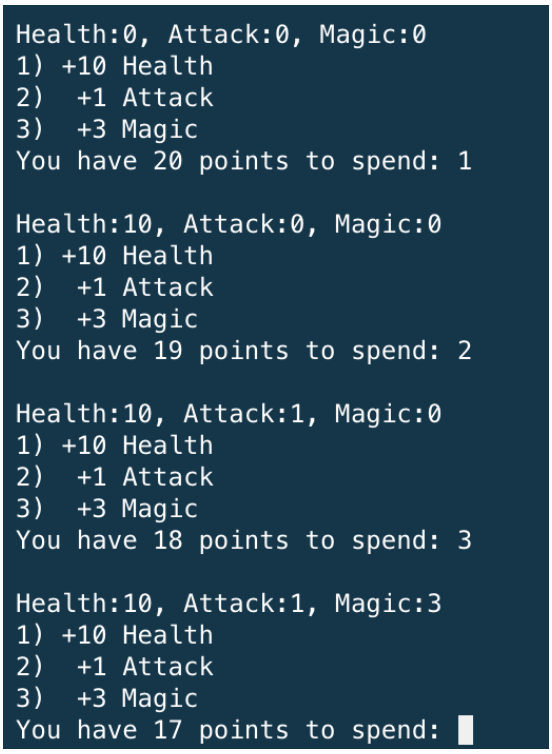
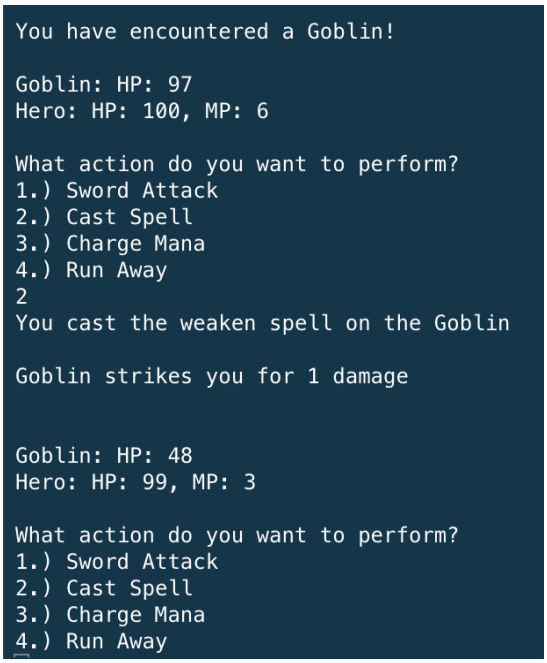
We will develop this project using an iterative design strategy while testing our code between each iteration. Your first iteration should start with implementing a method that creates a hero. The second iteration should implement a method that creates a monster. The final iteration should implement a method that manages the combat algorithm. This combat method may need to invoke sub-methods which handle the various attack options such as: melee actions, magic actions, charge actions, or flee actions. You are encouraged to customize, modify, or design your very own combat rules and stats so long as you meet the minimum number of options as listed within these specifications.

Required Concepts:

You will build a simple game application that uses all the fundamental concepts that we have covered up to Chapter 5. Concepts you may want to consider using for your game may include:

1. Named Constants (used for evaluating & executing user input)
2. Class variables (game data must be accessible to all methods in class)
3. Enumeration types (models the possible game states)
4. Class methods (used to break down the game logic into simpler parts)
5. Passing parameters (pass local data forward from one method to another)
6. Returning values (pass local data backward from one method to another)
7. Logical operators (evaluate multiple criteria for loop control)
8. Repetition statements (character creation and combat loop)
9. Random class (generate scoped random values for damage & health)

Expected Output (Final build):

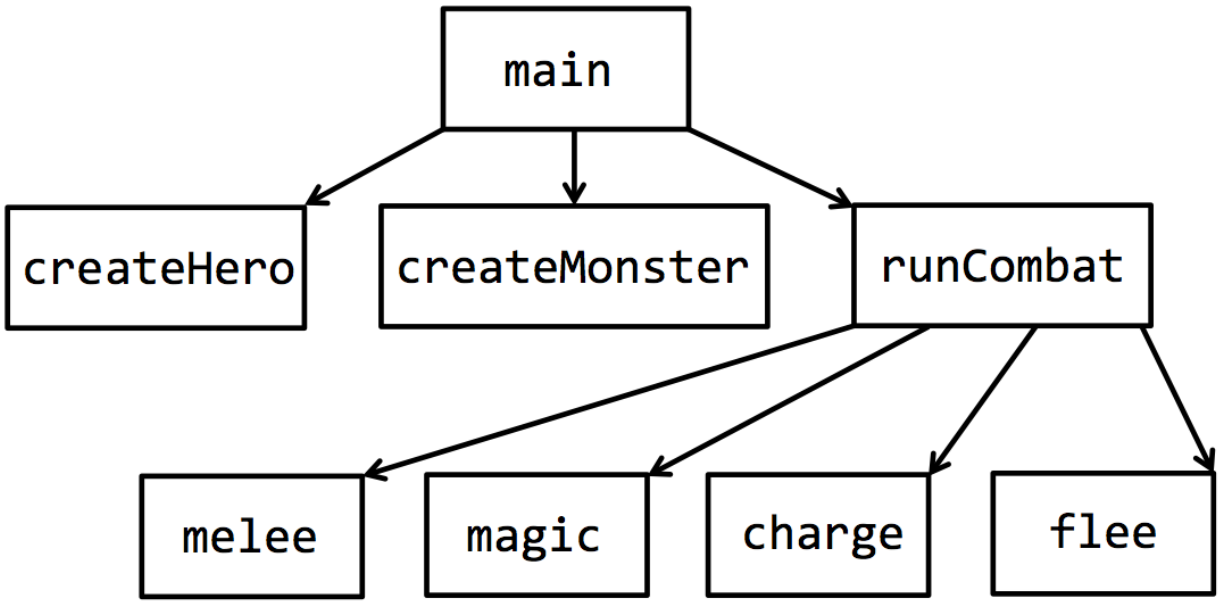
	A.) Example output of the Hero creation
	B.) Example output of the Monster Creation C.) Example output of the Combat loop

Top:

To start designing our game we first start with defining the Top. The Top is a single statement that defines the program's purpose or functionality.

```
A game where the player creates a character and uses it to battle random monsters.
```

Similar to the previous assignment we divide the *Top* into a series of smaller tasks and list them in the order in which they'll be performed. However, for this assignment we will use class methods to subdivide the algorithm into smaller, related blocks of instructions. We will continually create new methods until we have a complete, well-defined solution.



Example Application Hierarchy with methods

The main method should serve as the *manager* of your application and should delegate the actual work to other methods. We use the Top-down Stepwise refinement approach to identify the methods we need for our game. A major advantage of this approach, is that we may implement and test our game at each stage of the refinement process thereby allowing us to verify that the system works as we expect it should.

Refinement 1 of 3: **Hero Creation** **(25 points)**

In this refinement you must implement the *Hero Creation* functionality. The Hero creation should be an interactive process for the player to allow them to customize their own hero. The examples given in this document will default to the stats used in the previous project however you are free to implement this game with your own set of rules or stats. For this method, you should consider using the following:

1. Create new **class variables** to store the values of the hero's various stats. Initialize all of the hero's stats to 0. Your hero must have at least 3 stats. You may use the same stats given from the previous assignment:
    a. health
    b. attack power
    c. magic power

2. Create a new **class method** responsible for creating a hero. The Hero's creation should be an interactive process for the player. The player will be given a number of stat points, (e.g. 20 points) that they can individually spend to increase their hero's stats. While the player still has stat points remaining, the system should:
    a. Report to the user the hero's current stat values
    b. Display to the user what a stat point could purchase. Examples might include :
        i. 1 stat point gives +10 health
        ii. 1 stat point gives + 1 attack power
        iii. 1 stat point gives +3 magic power

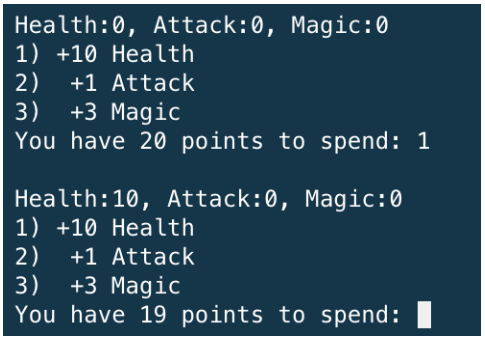
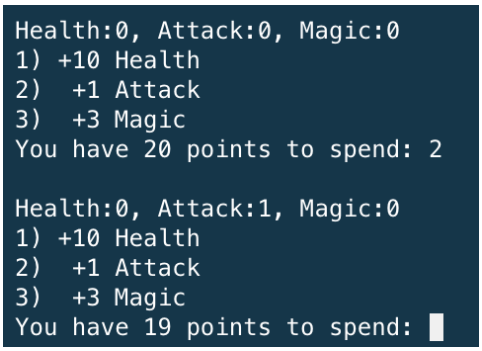
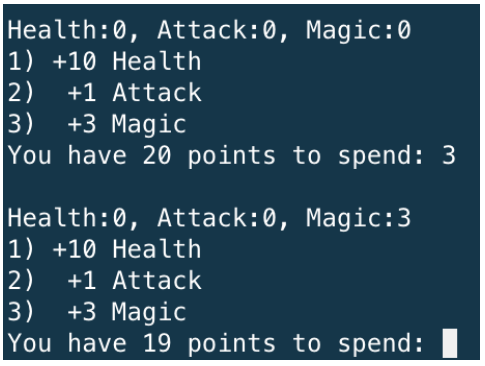
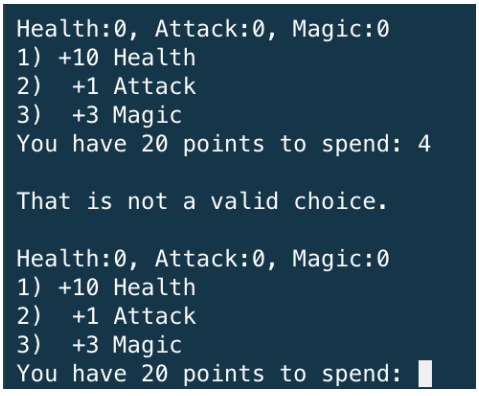
        Each of these options should have also have a corresponding integer indicating to the player how to select that choice. An example is given below:

        ```
        1.)     +10 health
        2.)     +1 attack power
        3.)     +3 magic power
        ```

    c. Report to the user how many stat points are left to spend
    d. Get the user's input
    e. Update the hero's variable that corresponds to the user's choice and decrement their remaining stat points. However, if the user didn't pick a valid option then report an error message and do **not** deduct a stat point.

3. Invoke this new *create hero* method from within the main method.

Expected Output (*Refinement 1: Hero Creation*):

	**Example 1 for the Hero Creation:** The initial values for the hero's variables are displayed as 0. The stat options are then listed. The player has 20 remaining stat points. They select option 1, to increase the hero' health. Since the hero still has stat points remaining then the loop continues. Notice that the hero's health is updated to 10 and the stat points is reduced to 19.
	**Example 2 for the Hero Creation:** The initial values for the hero's variables are displayed as 0. The stat options are then listed. The player has 20 remaining stat points. They select option 2, to increase the hero's attack power. Since the hero still has stat points remaining then the loop continues. Notice that the hero's attack power is updated to 1 and the stat points is reduced to 19.
	**Example 3 for the Hero Creation:** The initial values for the hero's variables are displayed as 0. The stat options are then listed. The player has 20 remaining stat points. They select option 3, to increase the hero's magic power. Since the hero still has stat points remaining then the loop continues. Notice that the hero's magic is updated to 3 and the stat points is reduced to 19.
	**Example 4 for the Hero Creation:** The initial values for the hero's variables are displayed as 0. The stat options are then listed. The player has 20 remaining stat points. They select an invalid option, 4. An error message prints to notify the player and no stat points are reduced.

Refinement 2 of 3: **Monster Creation** **(25 points)**

In this refinement you must implement the *Monster Creation* functionality. The monster creation should be a randomized process that generates different types of monsters. Your game should be able to generate at least 3 different types of monsters. The examples given in this document will default to the stats used in the the previous project however you are free to implement this game with your own set of rules or stats. For this portion of the game you may want to consider using the following:

1. Create new **class variables** to store the monster's various stats. Your monster should have at least three attributes. You may use the same stats given in the previous assignment:
    a. monster's name
    b. monster's health
    c. monster's attack power
    d. monster's experience point value (That is, the amount of experience points the player gets for defeating this monster).
        i. This attribute you may find useful if you do the bonus and allow the player to level up.
2. Create a new **class method** responsible for generating a random monster. Since monster's are defined by the values of their variables, we can accomplish this through a multi-selection statement. There must be a number of selections based on the number of different types of monsters that you have. Each option within the multi-selection statement sets the values for the monster class variables differently based on monster type. You must have a minimal of 3 types of monsters. In our test application, we created Goblins, Orcs, and Trolls and our method worked as following:
    a. Generate a random number. The scale of this random value should be based on the number of possible monsters.
    b. Use the random number as the control variable for a multi-selection statement to decide which type of monster to generate by setting the monster's variables such as its name, health, attack power, and experience point value.
        i. For a more interesting game the monster's health and attack power should contain a constant component and random component. For example, on the game that we designed we assigned the following values for our monsters:

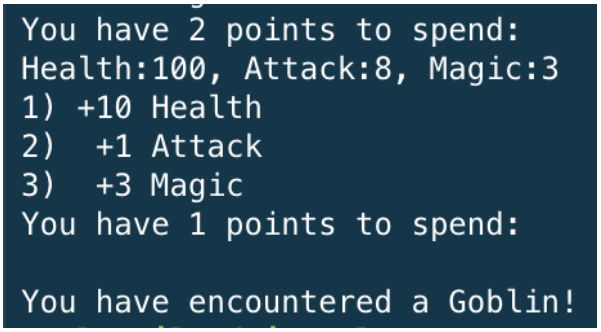
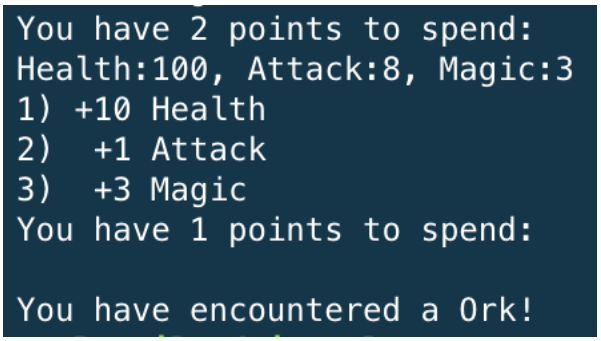
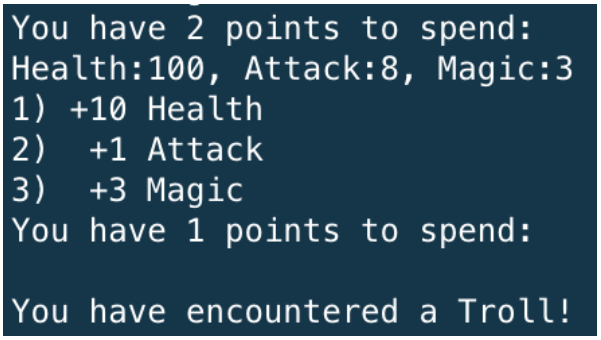
random	name	attack power	health	xp
0	"goblin"	8 + (0 to 4)	75 + (0 to 24)	1
1	"orc"	12 + (0 to 4)	100 + (0 to 24)	3
2	"troll"	15 + (0 to 4)	150 + (0 to 49)	5

example random monsters stats

    c. Report to the user that a monster has been created:
    `" You have encountered a <<monster's name>>`

3. Invoke the *create monster* method within the main method.

Expected Output (*Refinement 2: Monster Creation*):

	**Example 1 for the Monster Creation:** After the *hero creation* method completes the main method should invoke the *monster creation* method. On this execution the game has randomly generated a **Goblin monster**. This is reported to the user on the last line.
	**Example 2 for the Monster Creation:** After the *hero creation* method completes the main method should invoke the *monster creation* method. On this execution the game has randomly generated a **Ork monster**. This is reported to the user on the last line.
	**Example 3 for the Monster Creation:** After the *hero creation* method completes the main method should invoke the *monster creation* method. On this execution the game has randomly generated a **Troll monster**. This is reported to the user on the last line.

Refinement 3 of 3: **Managing Combat** **(50 points)**

In this refinement you must implement the *Combat System*. This may require you to create additional sub-methods to handle various tasks within combat. The algorithm used to model combat could be the same as the previous assignment. However, you are free to design and create your own combat options. You must have at least 4 options. The combat should loop until either the hero runs away, dies, or kills the monster. Consider implementing using the following strategy:

1. Create a new **class method** responsible for managing your combat algorithm. This method should handle the following responsibilities:
    a. Report the Hero's current status such as their health and magic points
    b. Report the Monster's current status such as its name and health
    c. Display a prompt with the combat options that user may select. You must provide at least four different combat options such as melee attack, magic attack, charge magic, flee combat. You are encouraged to design and create your own combat options.
    d. Use an input operation to get the user's selection
    e. Use a multi-selection operation to determine the results for the player's actions. A good design strategy for implementing the various combat actions is to define them within their own methods. Don't forget to report an error message if the user selects an invalid option.

2. Create new **class methods** responsible for processing your individual combat options. Examples from the previous project include:

    a. melee option
        i. Calculate the random damage by hero using a range from 1 to their attack power.
        ii. Reduce the monster's health by that amount of damage.
        iii. Report to the player the amount of damage you dealt to the monster.
        iv. If the monster is still alive:
            1. It strikes the hero for random damage based on its attack power
            2. Report to player that they have been attacked

    b. magic option
        i. If the hero has enough mana to cast the spell:
            1. Reduce the monster's health by half
            2. Report to player that you cast a spell on the monster
            3. If the monster is still alive:
                a. It strikes the hero for random damage based on its attack power
                b. Report to player that they have been attacked
        ii. If the hero does not have enough mana to cast the spell:
            1. Report that they can't do that because they don't have enough mana.

c. charge option

i. Increment the hero's magic power
ii. Report to the player that the hero charged their mana
iii. If the monster is still alive:
    1. It strikes the hero for random damage based on its attack power
    2. Report to player that they have been attacked

d. flee option

i. Toggle a loop control variable to exit the combat loop. Note that the flee option may be required to return a value to its invoker to control the loop control variable.
ii. Report to the player that they ran away

3. Invoke the *run combat* method within the main method using a loop. The loop should not terminate until either the hero has run away, is slain, or kills the monster.

Expected Output (*Refinement 3: Managing Combat*):

`` You have encountered a Ork! Ork: HP: 108 Hero: HP: 100, MP: 6 What action do you want to perform? 1.) Sword Attack 2.) Cast Spell 3.) Charge Mana 4.) Run Away 1 You attack the Ork for 6 damage Ork strikes you for 5 damage Ork: HP: 102 Hero: HP: 95, MP: 6 ```	**Example 1 for Managing Combat:** After the monster is created, combat begins. In this example the player selects to use a *sword attack*. The game reports that the player deals 6 damage to the Ork. The Ork then strikes the player for 5 damage. Notice in the next update that both the Monster and Hero health are both reduced by that amount of damage.
`` Ork: HP: 102 Hero: HP: 95, MP: 6 What action do you want to perform? 1.) Sword Attack 2.) Cast Spell 3.) Charge Mana 4.) Run Away 2 You cast the weaken spell on the Ork Ork strikes you for 1 damage Ork: HP: 51 Hero: HP: 94, MP: 3 ```	**Example 2 for Managing Combat:** After the monster is created, combat begins. In this example the player selects to *Cast Spell*. The game reports that the player has successfully cast the spell on the Ork. The Ork then strikes the player for 1 damage. Notice in the next update that Hero health is reduced by that amount of damage and the monster health is halved. Also notice that the player's magic points have been reduced by 3.
`` Ork: HP: 51 Hero: HP: 94, MP: 3 What action do you want to perform? 1.) Sword Attack 2.) Cast Spell 3.) Charge Mana 4.) Run Away 3 You focus and charge your mana. Ork strikes you for 10 damage ```	**Example 3 for Managing Combat:** After the monster is created, combat begins. In this example the player selects to *Charge Mana*. The game reports that the player has charged their magic. The Ork then strikes the player for 10 damage. Notice in the next update that Hero health is reduced by that amount of damage. Also notice that the player's magic points have been increased by 1.

<table>
<tr>
<td>

```
Ork: HP: 25
Hero: HP: 79, MP: 1

What action do you want to perform?
1.) Sword Attack
2.) Cast Spell
3.) Charge Mana
4.) Run Away
2
You don't have enough mana!


Ork: HP: 25
Hero: HP: 79, MP: 1
```

</td>
<td>

Example 4 for Managing Combat:
After the monster is created, combat begins. In this example the player selects to *Cast Spell*. The game reports that the player does not have enough mana to cast the spell. Notice in the next update that both the hero and monster stats remain the same.

</td>
</tr>
<tr>
<td>

```
Ork: HP: 25
Hero: HP: 79, MP: 1

What action do you want to perform?
1.) Sword Attack
2.) Cast Spell
3.) Charge Mana
4.) Run Away
5
That is not a valid option

Ork: HP: 25
Hero: HP: 79, MP: 1
```

</td>
<td>

Example 5 for Managing Combat:
After the monster is created, combat begins. In this example the player selects an *Invalid Option*. The game reports that to the player that is not a valid option. Notice in the next update that both the hero and monster stats remain the same.

</td>
</tr>
<tr>
<td>

```
Ork: HP: 25
Hero: HP: 79, MP: 1

What action do you want to perform?
1.) Sword Attack
2.) Cast Spell
3.) Charge Mana
4.) Run Away
4
You run away!

scalemailted@java-lectures:~/workspace/Homeworks $
```

</td>
<td>

Example 6 for Managing Combat:
After the monster is created, combat begins. In this example the player selects to *Run Away*. The game reports that the player ran away and then terminates the combat loop.

</td>
</tr>
</table>

Bonus:

Build a more complete game from this base skeletal structure. Loop this program to allow the player to continue fighting and track the hero's level. As the hero kills monsters level him up increasing his attack power and health.

Submissions:

You should create a new folder named "hw3" in your gitlab project, put your source files in that folder, and add, commit, and push those files to gitlab.