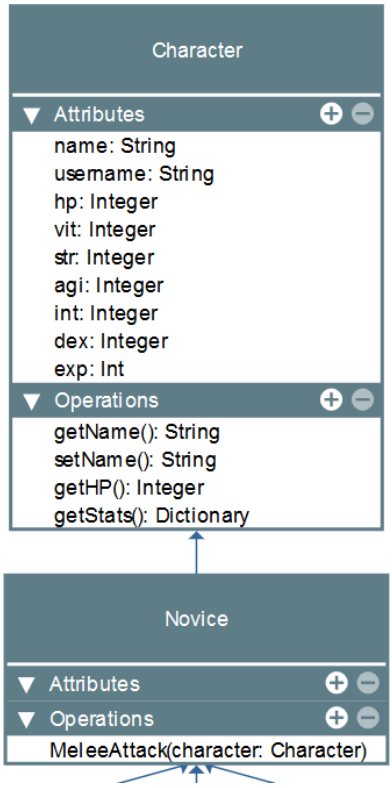
Laboratory Activity No. 2	
Inheritance, Encapsulation, and Abstraction	
Course Code: CPE009	**Program:** BSCPE
Course Title: Object-Oriented Programming	**Date Performed: 09/29/2024**
Section: CPE21S4	**Date Submitted: 09/29/2024**
Name: REYES, ALEXZANDER J.	**Instructor: Professor Maria Rizette Sayo**
1. Objective(s):	
This activity aims to familiarize students with the concepts of Object-Oriented Programming	
2. Intended Learning Outcomes (ILOs):	
The students should be able to: 2.1 Identify the possible attributes and methods of a given object 2.2 Create a class using the Python language 2.3 Create and modify the instances and the attributes in the instance.	
3. Discussion:	

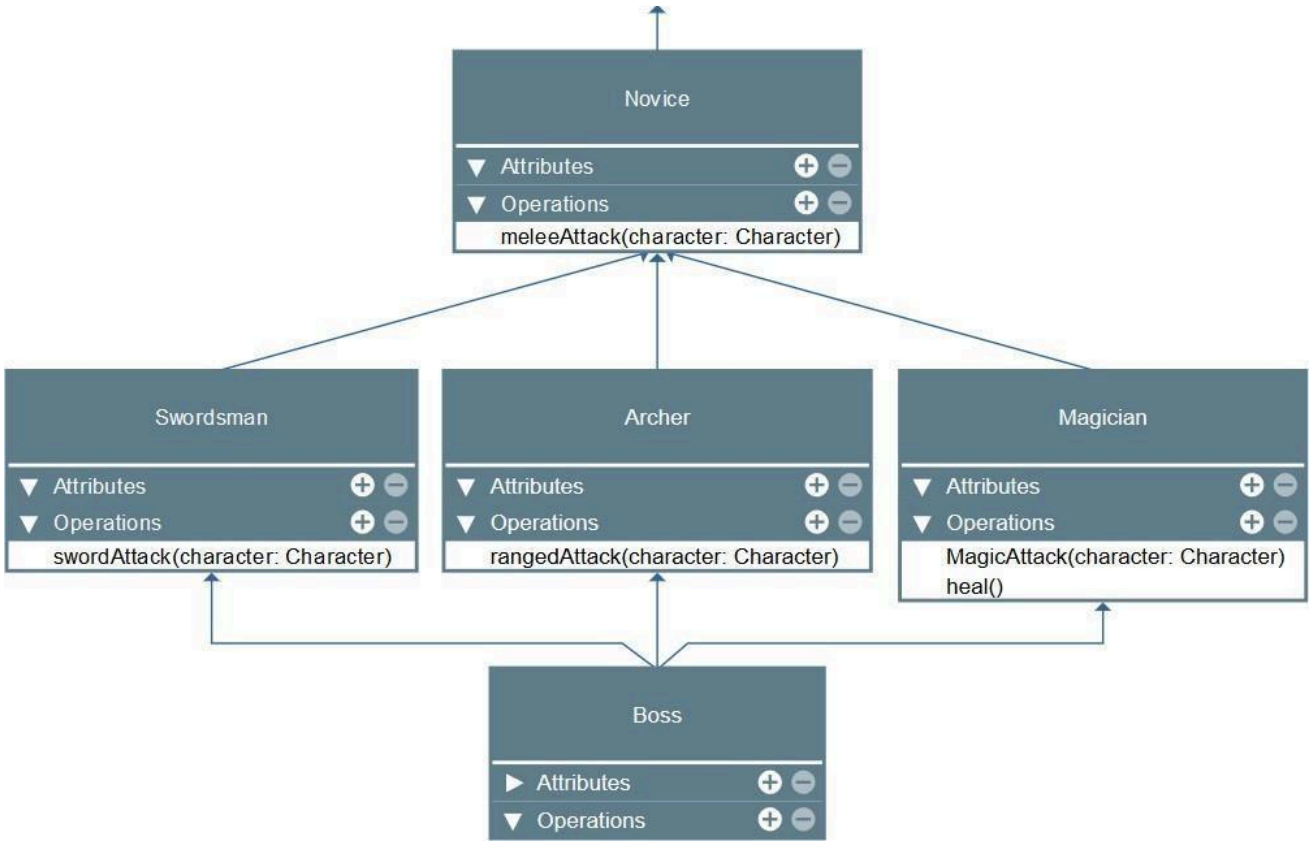
Object-Oriented Programming (OOP) has 4 core Principles: Inheritance, Polymorphism, Encapsulation, and Abstraction. The main goal of Object-Oriented Programming is code reusability and modularity meaning it can be reused for different purposes and integrated in other different programs. These 4 core principles help guide programmers to fully implement Object-Oriented Programming. In this laboratory activity, we will be exploring Inheritance while incorporating other principles such as Encapsulation and Abstraction which are used to prevent access to certain attributes and methods inside a class and abstract or hide complex codes which do not need to be accessed by the user.

An example is given below considering a simple UML Class Diagram:



The Base Character class will contain the following attributes and methods and a Novice Class will become a child of Character. The OOP Principle of Inheritance will make Novice have all the attributes and methods of the Character class as well as other

unique attributes and methods it may have. This is referred to as Single-level Inheritance. In this activity, the Novice class will be made the parent of three other different classes Swordsman, Archer, and Magician. The three classes will now possess the attributes and methods of the Novice class which has the attributes and methods of the Base Character Class. This is referred to as Multi-level inheritance.



The last type of inheritance that will be explored is the Boss class which will inherit from the three classes under Novice. This Boss class will be able to use any abilities of the three Classes. This is referred to as Multiple inheritance.

4. Materials and Equipment:

Desktop Computer with Anaconda
Python Windows Operating System

5. Procedure:

Creating the Classes

1. Inside your folder **oopfa1_<lastname>**, create the following classes on separate .py files with the file names: Character, Novice, Swordsman, Archer, Magician, Boss.
2. Create the respective class for each .py files. Put a temporary pass under each class created except in Character.py Ex.
   ```
   class Novice():
       pass
   ```
3. In the Character.py copy the following codes

```python
class Character():
    def __init__(self, username):
        self.__username = username
        self.__hp = 100
        self.__mana = 100
        self.__damage = 5
        self.__str = 0 # strength stat
        self.__vit = 0 # vitality stat
        self.__int = 0 # intelligence stat
        self.__agi = 0 # agility stat
    def getUsername(self):
        return self.__username
    def setUsername(self, new_username):
        self.__username = new_username
    def getHp(self):
        return self.__hp
    def setHp(self, new_hp):
        self.__hp = new_hp
    def getDamage(self):
        return self.__damage
    def setDamage(self, new_damage):
        self.__damage = new_damage
    def getStr(self):
        return self.__str
    def setStr(self, new_str):
        self.__str = new_str
    def getVit(self):
        return self.__vit
    def setVit(self, new_vit):
        self.__vit = new_vit
    def getInt(self):
        return self.__int
    def setInt(self, new_int):
        self.__int = new_int
    def getAgi(self):
        return self.__agi
    def setAgi(self, new_agi):
        self.__agi = new_agi
    def reduceHp(self, damage_amount):
        self.__hp = self.__hp - damage_amount
    def addHp(self, heal_amount):
        self.__hp = self.__hp + heal_amount
```

Note: The double underscore signifies that the variables will be inaccessible outside of the class.

4. In the same Character.py file, under the code try to create an instance of Character and try to print the username Ex.
   character1 = Character("Your Username") print(character1.username)
   print(character1.getUsername())
5. Observe the output and analyze its meaning then comment the added code.

Single Inheritance

1. In the Novice.py class, copy the following code.

```python
from Character import Character

class Novice(Character):
    def basicAttack(self, character):
        character.reduceHp(self.getDamage())
        print(f"{self.getUsername()} performed Basic Attack! -{self.getDamage()}")
```

2. In the same Novice.py file, under the code try to create an instance of Character and try to print the username Ex.
   character1 = Novice("Your Username")
   print(character1.getUsername())
   print(character1.getHp())
3. Observe the output and analyze its meaning then comment the added code.

Multi-level Inheritance

1. In the Swordsman, Archer, and Magician .py files copy the following codes for each file: Swordsman.py

```python
from Novice import Novice

class Swordsman(Novice):
    def __init__(self, username):
        super().__init__(username)
        self.setStr(5)
        self.setVit(10)
        self.setHp(self.getHp()+self.getVit())

    def slashAttack(self, character):
        self.new_damage = self.getDamage()+self.getStr()
        character.reduceHp(self.new_damage)
        print(f"{self.getUsername()} performed Slash Attack! -{self.new_damage}")
```

Archer.py

```python
from Novice import Novice
import random

class Archer(Novice):
    def __init__(self, username):
        super().__init__(username)
        self.setAgi(5)
        self.setInt(5)
        self.setVit(5)
        self.setHp(self.getHp()+self.getVit())

    def rangedAttack(self, character):
        self.new_damage = self.getDamage()+random.randint(0,self.getInt())
        character.reduceHp(self.new_damage)
        print(f"{self.getUsername()} performed Slash Attack! -{self.new_damage}")
```

Magician.py

```python
from Novice import Novice

class Magician(Novice):
    def __init__(self, username):
        super().__init__(username)
        self.setInt(10)
        self.setVit(5)
        self.setHp(self.getHp()+self.getVit())

    def heal(self):
        self.addHp(self.getInt())
        print(f"{self.getUsername()} performed Heal! +{self.getInt()}")

    def magicAttack(self, character):
        self.new_damage = self.getDamage()+self.getInt()
        character.reduceHp(self.new_damage)
        print(f"{self.getUsername()} performed Magic Attack! -{self.new_damage}")
```

2. Create a new file called Test.py and copy the codes below:

```python
from Swordsman import Swordsman
from Archer import Archer
from Magician import Magician


Character1 = Swordsman("Royce")
Character2 = Magician("Archie")
print(f"{Character1.getUsername()} HP: {Character1.getHp()}")
print(f"{Character2.getUsername()} HP: {Character2.getHp()}")
Character1.slashAttack(Character2)
Character1.basicAttack(Character2)
print(f"{Character1.getUsername()} HP: {Character1.getHp()}")
print(f"{Character2.getUsername()} HP: {Character2.getHp()}")
Character2.heal()
Character2.magicAttack(Character1)
print(f"{Character1.getUsername()} HP: {Character1.getHp()}")
print(f"{Character2.getUsername()} HP: {Character2.getHp()}")
```

3. Run the program Test.py and observe the output.
4. Modify the program and try replacing Character2.magicAttack(Character1) with Character2.slashAttack(Character1) then run the program again and observe the output.

Multiple Inheritance

1. In the Boss.py file, copy the codes as shown:

```python
from Swordsman import Swordsman
from Archer import Archer
from Magician import Magician

class Boss(Swordsman, Archer, Magician): # multiple inheritance
    def __init__(self, username):
        super().__init__(username)
        self.setStr(10)
        self.setVit(25)
        self.setInt(5)
        self.setHp(self.getHp()+self.getVit())
```

2. Modify the Test.py with the code shown below:

```
from Swordsman import Swordsman
from Archer import Archer
from Magician import Magician
from Boss import Boss

Character1 = Swordsman("Royce")
Character2 = Boss("Archie")
print(f"{Character1.getUsername()} HP: {Character1.getHp()}")
print(f"{Character2.getUsername()} HP: {Character2.getHp()}")
Character1.slashAttack(Character2)
Character1.basicAttack(Character2)
print(f"{Character1.getUsername()} HP: {Character1.getHp()}")
print(f"{Character2.getUsername()} HP: {Character2.getHp()}")
Character2.heal()
Character2.basicAttack(Character1)
Character2.slashAttack(Character1)
Character2.rangedAttack(Character1)
Character2.magicAttack(Character1)
print(f"{Character1.getUsername()} HP: {Character1.getHp()}")
print(f"{Character2.getUsername()} HP: {Character2.getHp()}")
```

3. Run the program Test.py and observe the output.

6. Supplementary Activity:

Task

Create a new file Game.py inside the same folder use the pre-made classes to create a simple Game where two players or one player vs a computer will be able to reduce their opponent's hp to 0.

Requirements:

1. The game must be able to select between 2 modes: Single player and Player vs Player. The game can spawn multiple matches where single player or player vs player can take place.
2. In Single player:
    - the player must start as a Novice, then after 2 wins, the player should be able to select a new role between Swordsman, Archer, and Magician.
    - The opponent will always be a boss named Monster.
3. In Player vs Player, both players must be able to select among all the possible roles available except Boss.
4. Turns of each player for both modes should be randomized and the match should end when one of the players hp is zero.
5. Wins of each player in a game for both the modes should be counted.

```
import random

# Base Character Class
class Character:
    def __init__(self, name, hp, attack_power):
        self.name = name
        self.hp = hp
        self.attack_power = attack_power

    def attack(self, opponent):
        damage = random.randint(0, self.attack_power)
        opponent.hp -= damage
        print(f"{self.name} attacks {opponent.name} for {damage} damage!")
        if opponent.hp <= 0:
            print(f"{opponent.name} has been defeated!")
            opponent.hp = 0

    def is_alive(self):
        return self.hp > 0


# Subclasses for different roles
class Novice(Character):
    def __init__(self, name):
        super().__init__(name, 50, 10)


class Swordsman(Character):
    def __init__(self, name):
        super().__init__(name, 70, 15)


class Archer(Character):
    def __init__(self, name):
```

```
        super().__init__(name, 50, 20)


class Magician(Character):
    def __init__(self, name):
        super().__init__(name, 40, 25)


# Boss Class
class Monster(Character):
    def __init__(self):
        super().__init__("Monster", 100, 12)


# Game Class to handle game flow
class Game:
    def __init__(self):
        self.player1_wins = 0
        self.player2_wins = 0
        self.player = None
        self.opponent = None
        self.mode = None

    def select_mode(self):
        print("Select Game Mode:\n1. Single Player\n2. Player vs Player")
        self.mode = input("Enter 1 or 2: ")
        if self.mode == "1":
            self.single_player_mode()
        elif self.mode == "2":
            self.player_vs_player_mode()
        else:
            print("Invalid input, try again.")
            self.select_mode()

    def choose_role(self, player_name):
        print("Choose your role:\n1. Novice\n2. Swordsman\n3. Archer\n4. Magician")
        role_choice = input(f"{player_name}, enter 1, 2, 3, or 4: ")
        if role_choice == "1":
            return Novice(player_name)
        elif role_choice == "2":
            return Swordsman(player_name)
        elif role_choice == "3":
            return Archer(player_name)
        elif role_choice == "4":
            return Magician(player_name)
        else:
            print("Invalid input, try again.")
            return self.choose_role(player_name)

    def single_player_mode(self):
        print("Single Player Mode: You vs Monster")
```

```
        self.player = Novice("Player")
        self.opponent = Monster()
        self.play_match()

    def player_vs_player_mode(self):
        print("Player 1: Choose your role")
        player1 = self.choose_role("Player 1")
        print("Player 2: Choose your role")
        player2 = self.choose_role("Player 2")
        self.play_match(player1, player2)

    def play_match(self, player1=None, player2=None):
        if self.mode == "1":  # Single Player Mode
            players = [self.player, self.opponent]
        else:  # Player vs Player Mode
            players = [player1, player2]

        random.shuffle(players)  # Randomize turns

        while all([p.is_alive() for p in players]):
            attacker = players[0]
            defender = players[1]
            attacker.attack(defender)
            players.reverse()  # Swap turns

        # End of match logic
        if self.mode == "1":  # Single Player
            if self.player.is_alive():
                print("You won!")
                self.player1_wins += 1
                if self.player1_wins >= 2:
                    print("You have won twice! You can now choose a new role.")
                    self.player = self.choose_role("Player")  # Choose new role only once after two wins
            else:
                print("You lost!")
        else:  # Player vs Player
            if player1.is_alive():
                print(f"{player1.name} wins!")
                self.player1_wins += 1
            else:
                print(f"{player2.name} wins!")
                self.player2_wins += 1

        self.play_again()

    def play_again(self):
        play_more = input("Do you want to play again? (y/n): ")
        if play_more.lower() == 'y':
            self.select_mode()
        else:
            print(f"Final score: Player 1 Wins: {self.player1_wins}, Player 2 Wins: {self.player2_wins}.")
```

```
        print("Thanks for playing!")


# Initialize the game
if __name__ == "__main__":
    game = Game()
    game.select_mode()
```

Output

Clear

```
Select Game Mode:
1. Single Player
2. Player vs Player
Enter 1 or 2: 1
Single Player Mode: You vs Monster
Player attacks Monster for 2 damage!
Monster attacks Player for 5 damage!
Player attacks Monster for 9 damage!
Monster attacks Player for 6 damage!
Player attacks Monster for 6 damage!
Monster attacks Player for 8 damage!
Player attacks Monster for 9 damage!
Monster attacks Player for 12 damage!
Player attacks Monster for 5 damage!
Monster attacks Player for 10 damage!
Player attacks Monster for 7 damage!
Monster attacks Player for 10 damage!
Player has been defeated!
You lost!
Do you want to play again? (y/n):
```

Questions

1. **Why is Inheritance important?**

   Inheritance is essential because it encourages code reuse, makes systems easier to maintain, supports extensibility, and logically organizes relationships between classes. It also helps reduce redundancy and keeps your codebase clean and manageable.

2. **Explain the advantages and disadvantages of using applying inheritance in an Object-Oriented Program.**

   Using inheritance in object-oriented programming offers several advantages, but it also comes with certain disadvantages. One major advantage is code reusability, where common functionality and attributes are defined in a parent class and reused across multiple child classes, reducing duplication and promoting cleaner, more maintainable code. Inheritance also provides a logical and hierarchical structure, allowing you to model relationships between entities, making the code easier to understand and extend. Additionally, it supports polymorphism, which enables writing flexible and general methods that can operate on parent class types but work with any child class, promoting versatility. However, inheritance has some drawbacks, such as the risk of tight coupling between parent and child classes, which can make changes in the parent class affect all its subclasses, potentially introducing bugs. It can also lead to an over-complicated hierarchy, where deep inheritance chains make code harder to follow and debug. Finally, inheritance can sometimes encourage inflexibility, as classes can only inherit from one parent in languages with single inheritance (like Java), which might limit functionality and lead to design issues. Properly balancing these factors is key to leveraging inheritance effectively.

---

3. Differentiate single inheritance, multiple inheritance, and multi-level inheritance.
   Single inheritance involves a child class inheriting from only one parent class, providing a straightforward relationship between the two (e.g., Class B inherits from Class A). Multiple inheritance allows a child class to inherit from more than one parent class, combining the attributes and behaviors of multiple classes into one (e.g., Class C inherits from both Class A and Class B), though it can lead to complexity such as the diamond problem. Multi-level inheritance occurs when a class inherits from another class, which itself is a child of a parent class, forming a chain (e.g., Class C inherits from Class B, and Class B inherits from Class A). This creates a hierarchy across multiple generations of classes.

4. **Why is super()._init_(username) added in the codes of Swordsman, Archer, Magician, and Boss?**

   The super().__init__(username) statement is used in the Swordsman, Archer, Magician, and Boss classes to call the __init__ method of their parent class, Character. This ensures that the child classes (like Swordsman, Archer, etc.) inherit and properly initialize the attributes defined in the parent class (Character), such as name, hp, and attack_power.

5. **How do you think Encapsulation and Abstraction helps in making good Object-Oriented Programs?**
   Together, Encapsulation and Abstraction help in creating well-structured programs by enforcing controlled access to data, simplifying interactions with objects, and promoting modularity. This leads to programs that are easier to debug, extend, and maintain, as each component of the system can be understood and worked on independently. These principles also reduce the risk of errors and improve the overall quality of software design.

7. Conclusion:

To sum up, the foundational principles of Object-Oriented Programming—Encapsulation and Abstraction significantly improve the caliber of program design. By limiting direct access to an object's properties and guaranteeing controlled interaction through clear protocols, encapsulation promotes data integrity and modularity while helping to protect an object's internal state. Conversely, abstraction reduces the complexity of systems by emphasizing the functions that an object is required to accomplish while hiding extraneous internal details. This enables developers to work with more straightforward interfaces and still have flexibility in how they implement the system.

8. Assessment Rubric: