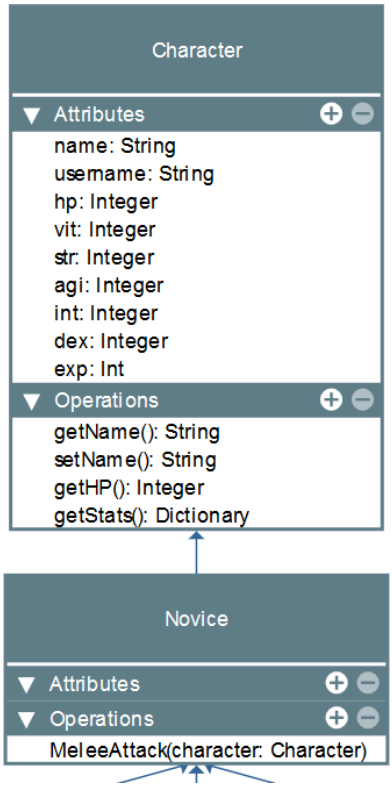
Laboratory Activity No. 2	
Inheritance, Encapsulation, and Abstraction	
Course Code: CPE009	**Program:** BSCPE
Course Title: Object-Oriented Programming	**Date Performed:**
Section:	**Date Submitted:**
Name:	**Instructor:**
1. Objective(s):	
This activity aims to familiarize students with the concepts 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:	

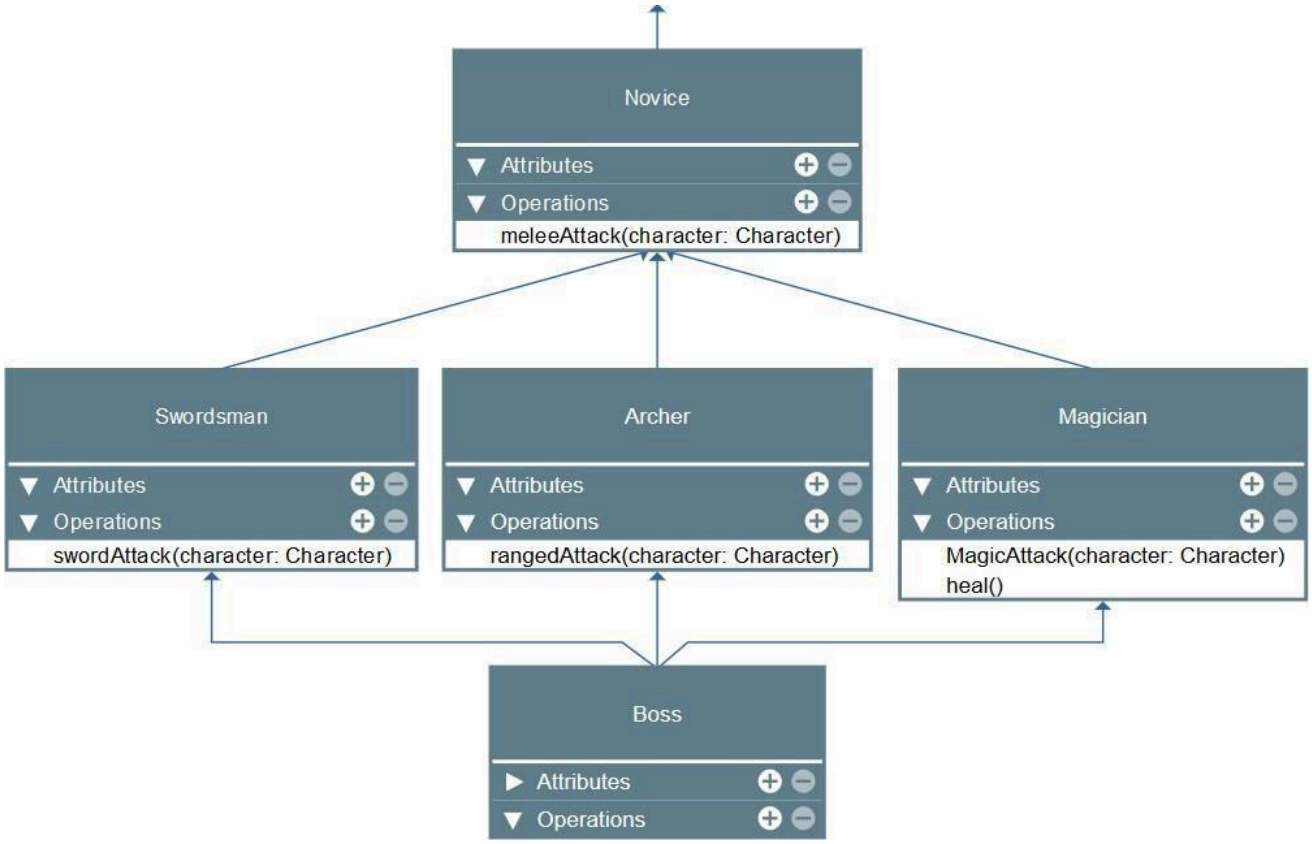
Object-Oriented Programming (OOP) has 4 core Principles: Inheritance, Polymorphism, Encapsulation, and Abstraction. The main goal of Object-Oriented Programming is code reusability and modularity meaning it can be reused for different purposes and integrated in other different programs. These 4 core 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 on 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 on 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 premade 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
from character import Character
from novice import Novice
from swordsman import Swordsman
from archer import Archer
from magician import Magician
from boss import Boss

class Game:
    NOVICE = 1
    SWORDSMAN = 2
    ARCHER = 3
    MAGICIAN = 4

    def __init__(self):
        self.player1 = None
        self.player2 = None
        self.current_player = None
        self.opponent = None
        self.game_mode = None
        self.player1_wins = 0
        self.player2_wins = 0

    def start(self):
        self.choose_game_mode()
        self.setup_players()
        self.game_loop()

    def choose_game_mode(self):
        while True:
            try:
                print("Select Game Mode:")
```

```
            print("1. Single Player")
            print("2. Player vs Player")
            choice = int(input("Enter your choice (1 or 2): "))
            if choice in (1, 2):
                self.game_mode = choice
                break
            else:
                print("Invalid choice. Please enter 1 or 2.")
        except ValueError:
            print("Invalid input. Please enter a number.")

    def setup_players(self):
        if self.game_mode == 1:
            self.player1 = self.get_class(self.NOVICE)
            self.player2 = Boss("Monster")
        else:
            self.player1 = self.select_player(1)
            self.player2 = self.select_player(2)

        self.current_player = random.choice([self.player1, self.player2])
        print(f"{self.current_player.getUsername()} starts first!")

    def select_player(self, player_num):
        while True:
            try:
                print(f"Player {player_num} Select Class:")
                print("1. Novice")
                print("2. Swordsman")
                print("3. Archer")
                print("4. Magician")
                choice = int(input("Enter your choice (1-4): "))
                if choice in (self.NOVICE, self.SWORDSMAN, self.ARCHER, self.MAGICIAN):
                    return self.get_class(choice)
                else:
                    print("Invalid choice. Please enter a number between 1 and 4.")
            except ValueError:
                print("Invalid input. Please enter a number.")

    def get_class(self, choice):
        username = input("Enter your username: ")
        if choice == self.NOVICE:
            return Novice(username)
        elif choice == self.SWORDSMAN:
            return Swordsman(username)
        elif choice == self.ARCHER:
            return Archer(username)
        elif choice == self.MAGICIAN:
            return Magician(username)

    def game_loop(self):
        while True:
```

```
            self.take_turn(self.current_player)
            if self.check_win():
                break
            self.switch_turns()
            if self.game_mode == 1:
                self.update_player1_after_wins()

    def take_turn(self, player):
        print(f"\n{player.getUsername()}'s Turn:")
        if self.game_mode == 1:
            player.attack(self.player2)
        else:
            self.opponent = self.get_opponent(player)
            player.attack(self.opponent)

        print(f"{player.getUsername()} HP: {player.getHp()}")
        print(f"{self.opponent.getUsername()} HP: {self.opponent.getHp()}")

    def get_opponent(self, player):
        return self.player2 if player == self.player1 else self.player1

    def switch_turns(self):
        self.current_player = self.get_opponent(self.current_player)

    def check_win(self):
        if self.player1.getHp() <= 0:
            self.player2_wins += 1
            print(f"{self.player2.getUsername()} Wins!")
            return True
        elif self.player2.getHp() <= 0:
            if self.game_mode == 1:
                self.player1_wins += 1
                print(f"{self.player1.getUsername()} Wins!")
            else:
                self.player1_wins += 1
                print(f"{self.player1.getUsername()} Wins!")
            return True
        return False

    def update_player1_after_wins(self):
        if self.player1_wins == 2:
            print("Congratulations! You've earned the right to choose a new class.")
            self.player1 = self.select_player(1)
```

Questions

1. Why is Inheritance important?
   Because inheritance improves the readability, maintainability, and modularity of code, it is significant. It is an essential element of OOP that promotes code organization and aids in the more effective construction of complicated systems.

2. Explain the advantages and disadvantages of applying inheritance in an Object-Oriented Program.
   While inheritance offers significant advantages like code reuse, maintainability, and flexibility through polymorphism, it can introduce complexity and tight coupling if overused or misapplied. Careful design decisions should be made about when to use inheritance versus composition or other OOP principles.

3. Differentiate single inheritance, multiple inheritance, and multi-level inheritance.
   In single inheritance, a class inherits from one parent class only. In multiple inheritance, a class can inherit from more than one parent class. Lastly, In multi-level inheritance, a class inherits from a parent class, which in turn inherits from another parent.

4. Why is super()._init_(username) added in the codes of Swordsman, Archer, Magician, and Boss?
   The super().__init__() function is crucial for ensuring proper initialization in inheritance scenarios, particularly when working with complex class hierarchies, which explains why the super()._init_ function was used for the said classes.

5. How do you think Encapsulation and Abstraction helps in making good Object-Oriented Programs?
   Encapsulation and Abstraction together help developers manage complexity, protect data, and write code that is modular, reusable, and easy to maintain, making them crucial components of good OOP design.

7. Conclusion:

In conclusion, the core ideas of OOP are inheritance, encapsulation, and abstraction. These ideas improve the modularity, organization, and maintainability of code. Although inheritance should be used sparingly to prevent needless complexity, it provides for code reuse and flexibility. While abstraction simplifies complex systems by concentrating on important features, encapsulation guarantees data safety and encourages modular architecture. When combined, these ideas support improved OOP development techniques by assisting in the creation of software systems that are effective, scalable, and maintainable.