Laboratory Activity # 2	
Introduction to Object-Oriented Programming	
Course Code: CPE009B	**Program:** BSCPE
Course Title: Object-Oriented Programming	**Date Performed:** 09-28-2024
Section: CPE21S4	**Date Submitted:** 09-28-2024
Name(s): Masangkay, Frederick D.	**Instructor:** Ma'am Maria Rizette Sayo
7. Supplementary Activity	

```
import random
from Archer import Archer
from Boss import Boss
from Novice import Novice
from Swordsman import Swordsman
from Magician import Magician

class Game:
    def __init__(self):
        self.single_player_wins = 0
        self.player_vs_player_wins = {"Player 1": 0, "Player 2": 0}
        self.roles = ["Swordsman", "Archer", "Magician"]

    def select_mode(self):
        mode = input("Select game mode (1 for Single Player, 2 for Player vs Player): ")
        if mode == "1":
            self.single_player_mode()
        elif mode == "2":
            self.player_vs_player_mode()
        else:
            print("Invalid mode. Please select a valid mode.")
            self.select_mode()

    def single_player_mode(self):
        player = Novice("Player")
        if self.single_player_wins >= 2:
            print("You have unlocked new roles!")
            role = input("Select a new role (Swordsman, Archer, Magician): ")
            if role == "Swordsman":
                player = Swordsman("Player")
            elif role == "Archer":
                player = Archer("Player")
            elif role == "Magician":
                player = Magician("Player")
            else:
                print("Invalid role. Defaulting to Novice.")
        opponent = Boss("Monster")
        self.play_match(player, opponent)
        self.select_mode()

    def player_vs_player_mode(self):
        player1 = self.select_role("Player 1")
        player2 = self.select_role("Player 2")
        self.play_match(player1, player2)
        self.select_mode()

    def select_role(self, player_name):
        print("Select a role for", player_name)
        for i, role in enumerate(self.roles):
            print(i+1, role)
        role_choice = input("Enter the number of your chosen role: ")
        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 role. Defaulting to Novice.")
            return Novice(player_name)

    def play_match(self, player1, player2):
        players = [player1, player2]
        random.shuffle(players)
        while player1.getHp() > 0 and player2.getHp() > 0:
            for player in players:
                print("\n", player.getUsername(), "turn")
                action = input("Enter 'attack' to attack or 'heal' to heal: ")
                if action == "attack":
                    if player == player1:
                        player1.basickAttack(player2)
                    else:
                        player2.basickAttack(player1)
                elif action == "heal":
                    if player == player1:
                        player1.addHp(10)  # heal 10 HP
                    else:
                        player2.addHp(10)  # heal 10 HP
                print(player1.getUsername(), "HP:", player1.getHp())
                print(player2.getUsername(), "HP:", player2.getHp())
        if player1.getHp() <= 0:
            print(player2.getUsername(), "wins!")
            if player2.getUsername() == "Player":
                self.single_player_wins += 1
            else:
                self.player_vs_player_wins[player2.getUsername()] += 1
        else:
            print(player1.getUsername(), "wins!")
            if player1.getUsername() == "Player":
                self.single_player_wins += 1
            else:
                self.player_vs_player_wins[player1.getUsername()] += 1
    def start_game(self):
        print("Welcome to the game!")
        self.select_mode()

game = Game()
game.start_game()
```

Questions

1. Why is Inheritance important?
    - Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows one class to inherit the properties and behavior of another class, enabling code reuse, facilitating the creation of a hierarchy of related classes, and promoting modularity and maintainability. By inheriting common attributes and methods from a parent class, child classes can build upon existing functionality, reducing code duplication and making it easier to modify or extend the program.
2. Explain the advantages and disadvantages of applying inheritance in an Object-Oriented Program.
    - Inheritance has both advantages and disadvantages. On the one hand, it allows for code reuse, easier maintenance, and improved readability. However, it can also create tight coupling between classes, making it difficult to change one class without affecting others, and can lead to the fragile base class problem and multiple inheritance complexity.
3. Differentiate single inheritance, multiple inheritance, and multi-level inheritance.
    - Single inheritance occurs when a child class inherits from a single parent class, while multiple inheritance occurs when a child class inherits from multiple parent classes. Multi-level inheritance, on the other hand, occurs when a child class inherits from a parent class that itself inherits from another parent class, creating a hierarchy of inheritance.
4. Why is super(). init (username) added in the codes of Swordsman, Archer, Magician, and Boss?
    - The super().__init__(username) line is used to call the constructor of the parent class (Character) from the child classes (Swordsman, Archer, Magician, and Boss), ensuring that the child classes inherit the attributes and behavior of the parent class, including the username attribute.
5. How do you think Encapsulation and Abstraction helps in making good Object-Oriented Programs?
    - Encapsulation and abstraction are essential principles of Object-Oriented Programming that help create robust and maintainable programs by hiding an object's internal state and behavior, and representing complex systems in a simplified way, respectively. This helps to protect the object's internal state, improve code organization and modularity, reduce complexity, and facilitate communication and collaboration among developers.

8. Conclusion

In conclusion, inheritance, encapsulation, and abstraction are fundamental concepts in Object-Oriented Programming that play a crucial role in creating robust and maintainable programs. Inheritance enables code reuse and facilitates the creation of a hierarchy of related classes, while encapsulation and abstraction help protect an object's internal state and simplify complex systems. By understanding and applying these concepts, developers can create efficient, modular, and scalable programs that are easy to maintain and extend.