PSMDSRC103 – PROGRAMMING
RAFFY BASAL

Part I

1. SINGLE INHERITANCE

In single inheritance, a class inherits from one and only one parent class. The child class can reuse code from the parent class. This is the simplest form of inheritance, allowing for an easy-to-understand class structure. You can extend and specialize the parent class without altering it. *(1:1 Relationship)*

2. MULTIPLE INHERITANCE

In multiple inheritance it can inherit from more than one parent class, meaning the child class gets the combined features and behaviors of all parent classes. Multiple inheritance allows you to combine the functionality of multiple classes into a single class. *(M:M Relationship)*

3. MULTILEVEL INHERITANCE

In multilevel inheritance it inherits from another class, which itself inherits from another parent class. In this type of inheritance, a class forms a chain-like hierarchy. Each level in the hierarchy can reuse and extend the properties and behaviors of its parent class. With each new level, the subclass can become more specific in behavior and attributes.
****(e.g. class animal as parent, class dog inherits from class parent, class Labrador inherits from class dog). (1:M Relationship)*

4. HIERARCHICAL INHERITANCE

In hierarchical inheritance, multiple child classes inherit from the same parent class. The parent class's functionality can be reused by multiple child classes while each child class can implement its own specific behavior. It helps in organizing classes in a structure where a common parent class shares its characteristics with multiple subclasses. *(1:M Relationship)*

5. POLYMORPHISM

Polymorphism allows different classes to be treated as instances of the same class through a common interface (usually via inheritance). This means that the same method can have different behaviors depending on which class it belongs to. It allows code to handle objects of different classes in a uniform way. This demonstrates how a single function can work in different ways based on the object passed to it.

PART II

Question No. 1

In this example, we set all the attributes into private by using double underscore "__". This means they cannot be accessed or modified directly from outside the class and you need to use getter and setter methods to interact with these private attributes.

GETTERS

Getters are methods that allow us to access the value of a private attribute. These getters provide a controlled way to retrieve the values of private attributes without exposing them directly. Remember the "get_" in using getters method.

e.g. get_student_id() returns the value of __student_id.

SETTERS

Setters are methods that allow us to modify the value of a private attribute. Setters give you control over how attributes are updated. You could, for example, add validation inside a setter to ensure the new value is valid (e.g., ensuring that age is not negative).

e.g. set_student_id() allows modifying the value of __student_id.

```python
# EXCERCISE No.1

class Student:
    def __init__(self, student_id, name, age, grade):
        # Private attributes
        self.__student_id = student_id
        self.__name = name
        self.__age = age
        self.__grade = grade

    # Getter for student_id
    def get_student_id(self):
        return self.__student_id

    # Setter for student_id
    def set_student_id(self, student_id):
        self.__student_id = student_id

    # Getter for name
    def get_name(self):
        return self.__name

    # Setter for name
    def set_name(self, name):
        self.__name = name

    # Getter for age
    def get_age(self):
        return self.__age

    # Setter for age
    def set_age(self, age):
        self.__age = age

    # Getter for grade
    def get_grade(self):
        return self.__grade

    # Setter for grade
    def set_grade(self, grade):
        self.__grade = grade
```

```python
# Example usage:
student = Student(101, "John Doe", 20, "A")

# Accessing private attributes using getters
print("ID:", student.get_student_id())
print("Name:", student.get_name())
print("Age:", student.get_age())
print("Grade:", student.get_grade())

print()
# Modifying private attributes using setters
student.set_name("RAFFY BASAL")
student.set_age(21)

# Display updated details
print("Updated Name:", student.get_name())
print("Updated Age:", student.get_age())

```

✓ 0.0s

```
ID: 101
Name: John Doe
Age: 20
Grade: A

Updated Name: RAFFY BASAL
Updated Age: 21
```

Question No. 2

We have an example here that showing a Hierarchical inheritance, having **multiple derived (child) classes** inherit from a **single base (parent) class.** In this case, both Undergrad and Graduate inherit from the same parent class, Student. Multiple child classes inheriting from a single parent class.

```python
# EXCERCISE No.2

# Base class Student
class Student:
    def __init__(self, student_id, name, age, grade):
        self.__student_id = student_id
        self.__name = name
        self.__age = age
        self.__grade = grade

    # Getter and setter methods
    def get_student_id(self):
        return self.__student_id

    def set_student_id(self, student_id):
        self.__student_id = student_id

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

    def get_age(self):
        return self.__age

    def set_age(self, age):
        self.__age = age

    def get_grade(self):
        return self.__grade

    def set_grade(self, grade):
        self.__grade = grade

    # Display basic student details
    def display_info(self):
        print(f"ID: {self.__student_id}, Name: {self.__name}, Age: {self.__age}, Grade: {self.__grade}")
```

```python
# Derived class Undergrad
class Undergrad(Student):
    def __init__(self, student_id, name, age, grade, major):
        super().__init__(student_id, name, age, grade)  # Inherit attributes from Student
        self.__major = major  # Additional attribute for undergrads

    # Getter and setter for major
    def get_major(self):
        return self.__major

    def set_major(self, major):
        self.__major = major

    # Display specific info for undergrad
    def display_info(self):
        super().display_info()
        print(f"Major: {self.__major}")

# Derived class Graduate
class Graduate(Student):
    def __init__(self, student_id, name, age, grade, research_topic):
        super().__init__(student_id, name, age, grade)  # Inherit attributes from Student
        self.__research_topic = research_topic  # Additional attribute for graduates

    # Getter and setter for research topic
    def get_research_topic(self):
        return self.__research_topic

    def set_research_topic(self, research_topic):
        self.__research_topic = research_topic

    # Display specific info for graduate
    def display_info(self):
        super().display_info()
        print(f"Research Topic: {self.__research_topic}")
```

```python
# Example usage:
undergrad_student = Undergrad(201, "Alice Smith", 19, "A", "Computer Science")
graduate_student = Graduate(301, "Bob Johnson", 25, "A", "Machine Learning")

# Displaying details for both students
print("Undergrad Student Info:")
undergrad_student.display_info()

print("\nGraduate Student Info:")
graduate_student.display_info()
```

✓ 0.0s

```
Undergrad Student Info:
ID: 201, Name: Alice Smith, Age: 19, Grade: A
Major: Computer Science

Graduate Student Info:
ID: 301, Name: Bob Johnson, Age: 25, Grade: A
Research Topic: Machine Learning
```

Question No. 3

The inheritance now in this example is **Multilevel Inheritance** because the Doctorate class inherits from Graduate, and Graduate inherits from Student. The chain of inheritance is: Student → Graduate → Doctorate.

```python
# Derived class for Masters students
class Masters(Graduate):
    def __init__(self, student_id, name, age, grade, research_topic, dissertation_title):
        super().__init__(student_id, name, age, grade, research_topic)
        self.__dissertation_title = dissertation_title  # New attribute specific to doctorate students

    def get_dissertation_title(self):
        return self.__dissertation_title

    def set_dissertation_title(self, dissertation_title):
        self.__dissertation_title = dissertation_title

    # Overriding display_info method to include dissertation details
    def display_info(self):
        super().display_info()
        print(f"Dissertation Title: {self.__dissertation_title}")

# Example usage:
Masters_student = Masters(401, "Charlie Green", 30, "A", "Artificial Intelligence", "Deep Learning in AI")

# Displaying the details of the doctorate student
print("Master's Student Info:")
Masters_student.display_info()
```

✓ 0.0s

```
Master's Student Info:
ID: 401, Name: Charlie Green, Age: 30, Grade: A
Research Topic: Artificial Intelligence
Dissertation Title: Deep Learning in AI
```

Polymorphism is demonstrated through **method overriding.** Both the Graduate and Masters classes override the display_info() method from their parent classes, this added new attribute of dissertation_tittle.

Each class has its own version of this method, tailored to its specific needs:

- The Graduate class includes the research topic.
- The Masters class includes both the research topic and the dissertation title.

This allows the display_info() method to behave differently based on the object type (either Graduate or Masters), even though it’s called in the same way.

Question No.4

Populating the Masters with three examples, the program will display the details of each Master's student before the deletion.

After deletion After deletion of master1, trying to call methods for master1 will raise a “NameError” because the object has been removed. The remaining instances (master2 and master3) will continue to work fine after the deletion of master1.

```python
# Graduate class (from previous example)
class Graduate(Student):
    def __init__(self, student_id, name, age, grade, research_topic):
        super().__init__(student_id, name, age, grade)
        self.__research_topic = research_topic

    def get_research_topic(self):
        return self.__research_topic

    def set_research_topic(self, research_topic):
        self.__research_topic = research_topic

    def display_info(self):
        super().display_info()
        print(f"Research Topic: {self.__research_topic}")

# Derived class for Master's students
class Masters(Graduate):
    def __init__(self, student_id, name, age, grade, research_topic, thesis_title):
        super().__init__(student_id, name, age, grade, research_topic)
        self.__thesis_title = thesis_title  # Attribute specific to Master's students

    def get_thesis_title(self):
        return self.__thesis_title

    def set_thesis_title(self, thesis_title):
        self.__thesis_title = thesis_title

    # Override display_info to include thesis title
    def display_info(self):
        super().display_info()
        print(f"Thesis Title: {self.__thesis_title}")

# Create 3 instances of the Master's class
master1 = Masters(501, "David White", 24, "A", "Data Science", "Big Data Analytics")
master2 = Masters(502, "Emily Clark", 25, "B", "Cybersecurity", "Network Vulnerabilities")
master3 = Masters(503, "Michael Brown", 26, "A", "Artificial Intelligence", "Neural Networks")
```

```
Master 1 Info:
ID: 501, Name: David White, Age: 24, Grade: A
Research Topic: Data Science
Thesis Title: Big Data Analytics

Master 2 Info:
ID: 502, Name: Emily Clark, Age: 25, Grade: B
Research Topic: Cybersecurity
Thesis Title: Network Vulnerabilities

Master 3 Info:
ID: 503, Name: Michael Brown, Age: 26, Grade: A
Research Topic: Artificial Intelligence
Thesis Title: Neural Networks

Error: name 'master1' is not defined

Master 2 Info (after deletion of master1):
ID: 502, Name: Emily Clark, Age: 25, Grade: B
Research Topic: Cybersecurity
Thesis Title: Network Vulnerabilities

Master 3 Info (after deletion of master1):
ID: 503, Name: Michael Brown, Age: 26, Grade: A
Research Topic: Artificial Intelligence
Thesis Title: Neural Networks
```

LEARNING AND CONCLUSION:

In object-oriented programming (OOP), **encapsulation** and the use of **getters and setters** provide control over how the data within a class is accessed and modified. By making class attributes private and exposing them through public methods, we can protect sensitive information and maintain the integrity of data. Getters retrieve values, while setters modify them, offering a controlled interface that ensures proper validation and restrictions when needed. Encapsulation enhances code security, reduces complexity, and improves maintainability, making it a fundamental principle in designing robust systems.

Inheritance, particularly **hierarchical and multilevel inheritance**, allows for code reuse by enabling classes to inherit properties and behaviors from other classes. By organizing related classes in parent-child relationships, we can avoid redundancy and promote cleaner code. For example, the Undergrad, Graduate, and Doctorate classes all extend the base Student class, each adding unique attributes and behaviors, while still sharing common functionality. This shows how inheritance streamlines development and creates modular, extendable codebases.

Lastly, **polymorphism** introduces flexibility by allowing objects of different classes to be treated uniformly through shared interfaces or method overriding. In our example, both Graduate and Doctorate classes override the display_info() method to display specific details about each type of student. This allows different objects to be used interchangeably while exhibiting distinct behaviors. Together, encapsulation, inheritance, and polymorphism form the core principles of OOP, enabling developers to create scalable, maintainable, and efficient software solutions.