Урок 7. Принципы ООП. Инкапсуляция и Наследование

ФИО_____

Тест:

1. *Чем отличается _name от __name в контексте атрибута класса?*
   A) Ничем B) __name преобразовывается в _ИмяКласса__name C) _name запрещает полный доступ извне D) __name удаляется при импорте
2. *Опишите своими словами, что из себя представляет **self** внутри класса?*

   _____
3. *Чем статический метод отличается от метода класса?*

   _____
4. *Что такое атрибуты и методы класса?*

   _____
5. *Что выведет данная программа?*

```
1    class Dog:  1 usage
2        def __init__(self, name, age):
3            self.__name = name
4            self._age = age
5
6    my_dog = Dog( name: "Buddy",  age: 3)
7    print(my_dog._age)
8    print(my_dog.__name)
```

Теория:

Принципы ООП:

- **Инкапсуляция:** «прячем детали, даём аккуратный интерфейс»

Этот принцип призван защитить внутреннее состояние объекта от случайного (или намеренного) неправильного использования.

Способы защиты данных нижними подчеркиваниями:

- Приватные атрибуты: начинаются с двойного подчеркивания __. Они не доступны напрямую из вне класса. Это достигается за счет механизма, называемого "name mangling", который изменяет имя атрибута, добавляя к нему имя класса.

```
class SafeBox:
    def __init__(self, pin):
        self.__pin = pin    # станет _SafeBox__pin

    def check(self, pin) -> bool:
        return pin == self.__pin
```

Технически доступно как obj._SafeBox__pin, так что это защита «от случайностей», не от злонамеренных действий. При попытке обратится к атрибуту **__pin** классически через экземпляр класса будет выдавать ошибку.

- Защищенные атрибуты: начинаются с одного подчеркивания _. Они доступны, но считается, что их не следует изменять напрямую. Это больше соглашение, чем строгий запрет, и оно помогает разработчикам понимать, что эти атрибуты предназначены для внутреннего использования.

```
class BankAccount:
    def __init__(self, owner):
        self.owner = owner
        self._balance = 0       # внутренний атрибут (convention)

    def deposit(self, amount: int):
        if amount <= 0:
            raise ValueError("amount>0")
        self._balance += amount

    def get_balance(self) -> int:
        return self._balance
```

Геттеры (чтение) и сеттеры (изменение), как способ защиты данных используются для управления доступом к приватным атрибутам. Они позволяют контролировать, как данные читаются и изменяются. Это особенно полезно, когда необходимо добавить проверку или логику при чтении или записи данных.

В Python можно использовать декоратор **@property** для создания геттеров и сеттеров, что делает код более элегантным и читаемым. Свойства позволяют определить методы, которые будут вызываться при доступе к атрибутам, что делает код более интуитивно понятным.

```
class Temperature:
    def __init__(self, celsius: float):
        self._c = celsius

    @property
    def celsius(self) -> float:
        return self._c # ГЕТТЕР

    @celsius.setter
    def celsius(self, value: float):
        if value < -273.15:
            raise ValueError("ниже абсолютного нуля")
        self._c = value # СЕТТЕР

    @property
    def fahrenheit(self) -> float:
        return self._c * 9/5 + 32    # ГЕТТЕР read-only: сеттера нет
```

Пример инкапсуляции:

```
class BankAccount:
    def __init__(self, initial_balance):
        self.__balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount

    @property
    def balance(self):
        return self.__balance

account = BankAccount(100)
account.deposit(50)
print(account.balance)  # 150
account.withdraw(30)
print(account.balance)  # 120
```

- **Наследование:** расширение поведение «родительского» класса через «дочерний».

Наследование позволяет создавать новые классы на основе уже существующих. Подкласс или (по-другому) Дочерний класс *расширяет* или *переопределяет* поведение базового.

Наследование (Inheritance): механизм, позволяющий одному классу (подклассу) унаследовать свойства и методы другого класса (родительского класса). Наследование способствует повторному использованию кода и упрощает его поддержку.

Родительский класс (Parent Class): класс, от которого наследуется другой класс. Родительский класс может содержать общие свойства и методы, которые будут унаследованы подклассами.

Подкласс (дочерний) (Subclass): класс, который наследует свойства и методы родительского класса. Подклассы могут добавлять новые свойства и методы, а также переопределять унаследованные.

Переопределение (Overriding): процесс замены метода родительского класса в подклассе. Переопределение позволяет изменять поведение унаследованных методов в подклассе.

```
class ParentClass:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, my name is {self.name}")

class ChildClass(ParentClass):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def greet(self):
        super().greet()
        print(f"I am {self.age} years old")
```

ChildClass наследует **ParentClass**. В его методе __init__ используется super(), чтобы вызвать метод __init__ родительского класса. Это позволяет инициализировать свойства родительского класса в подклассе. Метод greet переопределен, но также вызывает метод greet родительского класса с помощью **super()**.

Задачи:

1. **Инкапсуляция:**

- Создайте класс Rectangle, который имеет приватные атрибуты – ширину и высоту прямоугольника. Воспользуйтесь сеттером, чтобы передавать и изменять приватные параметры класса, к результирующему методу доступ не приватный. Результаты: площадь и периметр.
- Создайте класс Student, который имеет приватный атрибут – оценку, а также имя студента и два неприватных метода для добавления оценок и расчета среднего балла. Добавьте геттер для доступа к оценке.

2. **Наследование**

- Создайте родительский класс Person с атрибутами name и age, а также методом introduce, который выводит информацию о человеке.
  К нему создайте подкласс Student, который наследует Person и добавляет атрибут student_id. Переопределите метод introduce, чтобы он также выводил student_id.
  Также создайте подкласс Teacher, который наследует Person и добавляет атрибут subject. Переопределите метод introduce, чтобы он также выводил subject.
- Создайте подкласс GraduateStudent, который наследует Student и добавляет атрибут thesis_topic. Переопределите метод introduce, чтобы он также выводил thesis_topic.
  Создайте еще один подкласс Principal, который наследует Teacher и добавляет атрибут years_of_experience. Переопределите метод introduce, чтобы он также выводил years_of_experience.
  Создайте класс Course, который содержит список студентов и метод для добавления студентов в курс. Реализуйте метод для вывода списка студентов в курсе.