InnoDom

Принципы ООП

Принципы ООП

Наследование

один из принципов ООП, который позволяет создавать **новые классы** на основе уже **существующих**, заимствуя их свойства и методы. Для того, чтобы класс наследовал от другого класса, нужно при его создании указать в скобках класс, от которого наследуется.

Наследование

```python
class Animal:
    def __init__(self):
        pass

    def eat(self):
        print('Амнямнямням')

class Cat(Animal):
    def __init_(self):
        super().__init__()

    def mrr(self):
        print('mrrrrrrrrccr')


cat1 = Cat()
cat1.eat()
cat1.mrr()
```

Класс-родитель – *Animal*, такие классы положено называть **суперклассом** или базовым классом. **Класс-потомок** – *Cat*, наследует все методы, а свойства может наследовать только если в конструкторе потомка определим с помощью функции **super** переменные, которые можем наследовать.

На примере видно, что класс-наследник может вызывать метод класса-родителя. Благодаря наследованию можно определить *общие методы* и *свойства* от родителей к наследникам, поэтому можно не переписывать одинаковые методы в 2 класса.

Множественное наследование

```python
class A:
    def method_a(self):
        print("Method A")

class B:
    def method_b(self):
        print("Method B")

class C(A, B):
    def method_c(self):
        print("Method C")


c1 = C()
c1.method_c()
c1.method_a()
c1.method_b()
```

Множественное наследование позволяет наследовать класс от **двух** и **более** родителей.

Такой вид наследования позволяет создавать и проектировать *сложные структуры* классов, связанных между собой, но его использование должно быть продуманным и аккуратным.

Множественное наследование

Если в классах-родителях методы называются **одинаково**, тогда вызывается метод **первого** передаваемого в скобках класса.

```
class A:
    def method(self):
        print("Method A")

class B:
    def method(self):
        print("Method B")

class C(A, B):
    def method_c(self):
        print("Method C")


c1 = C()
c1.method_c()
c1.method()
```

Множественное наследование

Для того, чтобы избежать такую проблему, можно использовать в потомке методы и указать классы **напрямую** или **избегать одинаковых названий** методов.

```
class A:
    def method(self):
        print("Method A")

class B:
    def method(self):
        print("Method B")

class C(A, B):
    def method_c(self):
        A.method(self)
        B.method(self)


c1 = C()
c1.method_c()
```

Инкапсуляция

ещё один принцип ООП, который позволяет **скрыть** атрибуты либо методы от общего использования в классе.
Инкапсуляция позволяет использовать такие методы и атрибуты **только при реализации класса**. Это делается для методов и свойств, которые нужны в использовании только **внутри класса**.

Инкапсуляция

```python
class MyClass:
    def __init__(self):
        self.__private_attribute = 42

    def __private_method(self):
        print("This is a private method.")

    def public_method(self):
        print("This is a public method.")
        self.__private_method()
        print(self.__private_attribute)


my_object = MyClass()
my_object.public_method()
```

Поля либо методы скрываются, когда перед названием стоит **два нижних подчеркивания**.

! К скрытым методам либо полям **нельзя** обратится вне класса.

```python
class Robot:
    def __init_(self):
        self.__mood = 50
        self.__energy = 50

    def __charge(self):
        self.__energy = 100
        print("Робот заряжается")

    def play(self):
        print(self.__energy)
        print(self.__mood)
        if self.__energy >= 50:
            print("Играем")
            self.__mood += 10
            self.__energy -= 50
        else:
            self.__charge()

    def feed(self):
        print("Заряжаемся")
        self.__energy += 30
```

Внутренним состоянием робота являются **частные** (private) переменные: *настроение*(mood) и *энергия* (energy). Он также имеет частный(private) метод зарядки *charge()*. Робот может вызвать его в любой момент, когда захочет, другие классы не могут говорить роботу, когда ему нужно зарядится.
То, что им можно делать, определяется в **публичных** (public) методах *play()* и *feed()*. Каждый из них каким-то образом влияет на внутреннее состояние робота и может вызвать *charge()*. Таким образом, устанавливается связь между внутренним состоянием объекта и публичными методами.

```python
robot = Robot()
robot.play()
robot.play()
robot.feed()
```