

# Объектно-ориентированное программирование

## Лекция 3. Инкапсуляция и Наследование в Python

Определение и назначение инкапсуляции. Скрытие данных, интерфейс. Модификаторы доступа в Python. `@property`. Наследование: базовые и производные классы. Множественное наследование, MRO, `super()`.

1. Абстракция
2. Инкапсуляция
3. Наследование
4. Полиморфизм



**Абстракция** - принцип ООП, согласно которому объект характеризуется свойствами, которые отличают его от всех остальных объектов и при этом четко определяют его концептуальные границы.



**Абстракция** - принцип ООП, согласно которому объект характеризуется свойствами, которые отличают его от всех остальных объектов и при этом четко определяют его концептуальные границы.

1. Выделить главные и наиболее значимые свойства предмета.



**Абстракция** - принцип ООП, согласно которому объект характеризуется свойствами, которые отличают его от всех остальных объектов и при этом четко определяют его концептуальные границы.

1. Выделить главные и наиболее значимые свойства предмета.
2. Отбросить второстепенные характеристики.

***Инкапсуляция** - принцип ООП, согласно которому сложность реализации программного компонента должна быть спрятана за его интерфейсом.*

***Инкапсуляция** - принцип ООП, согласно которому сложность реализации программного компонента должна быть спрятана за его интерфейсом.*

1. Отсутствует доступ к внутреннему устройству программного компонента.
2. Взаимодействие компонента с внешним миром осуществляется посредством интерфейса, который включает публичные методы и поля.

***Инкапсуляция** - принцип ООП, согласно которому сложность реализации программного компонента должна быть спрятана за его интерфейсом.*

1. Отсутствует доступ к внутреннему устройству программного компонента.
2. Взаимодействие компонента с внешним миром осуществляется посредством интерфейса, который включает публичные методы и поля.

**Для чего нужна инкапсуляция?**

***Инкапсуляция** - принцип ООП, согласно которому сложность реализации программного компонента должна быть спрятана за его интерфейсом.*

1. Отсутствует доступ к внутреннему устройству программного компонента.
2. Взаимодействие компонента с внешним миром осуществляется посредством интерфейса, который включает публичные методы и поля.

### **Для чего нужна инкапсуляция?**

1. Инкапсуляция упрощает процесс разработки.
2. Повышается надежность программ.
3. Становится более легким обмен компонентами между программами.



# Модификаторы доступа

1. `public` - доступны для чтения и записи откуда угодно
2. `protected` - доступны для использования внутри класса и для потомков
3. `private` - доступны для использования только внутри класса

*В Python нет строгих `private` или `protected`, как в Java/C++. Вместо этого используются соглашения об именовании.*

Атрибуты без подчеркиваний. Доступны для чтения и записи откуда угодно. Это стандартное поведение.

```
1 class User:
2     def __init__(self, name):
3         self.name = name # Публичный атрибут
4
5 user = User("Alice")
6 print(user.name)         # OK
7 user.name = "Bob"       # OK
```

Атрибуты, начинающиеся с одного подчеркивания ( `_` ). Это сигнал для других разработчиков: "Я — внутренняя часть реализации этого класса. Не трогайте меня напрямую извне, если не уверены в своих действиях".

*Python никак не ограничивает доступ к таким атрибутам. Это просто соглашение.*

```
1 class BankAccount:
2     def __init__(self, amount):
3         self._balance = amount # "Защищенный" атрибут
4
5 acc = BankAccount(1000)
6 print(acc._balance) # Технически возможно, но считается плохой практикой
```

Атрибуты, начинающиеся с двух подчеркиваний ( `__` ). Python **изменяет (искажает)** имя такого атрибута, чтобы избежать случайных конфликтов имен в дочерних классах. Имя `__balance` внутри класса `BankAccount` превратится в `_BankAccount__balance`.

*Это не настоящая приватность, а механизм предотвращения коллизий. Зная новое имя, к атрибуту все еще можно получить доступ.*

```
1 class BankAccount:
2     def __init__(self, amount):
3         self.__balance = amount # "Приватный" атрибут
4
5 acc = BankAccount(1000)
6 # print(acc.__balance) # Вызовет AttributeError
7 print(acc._BankAccount__balance) # Так сработает, но делать так не нужно
```



```
1 class Person:
2     def __init__(self, age):
3         self.age = age
4
5 p1 = Person(18)
```



```
1 class Person:
2     def __init__(self, age):
3         self.age = age
4
5 p1 = Person(18)
6 p2 = Person(-1)
7 p3 = Person("twenty")
```



```
1 class Person:
2     def __init__(self, age):
3         self.set_age(age)
4
5     def set_age(self, age):
6         if not isinstance(age, int):
7             raise TypeError()
8         if not 0 <= age < 150:
9             raise ValueError()
10
11         self.age = age
12
13
14 p1 = Person(18)
15 p2 = Person(-1)
16 p3 = Person("twenty")
```



## Getter (Получение значения)

Декоратор `@property` над методом превращает его в атрибут, доступный только для чтения.

```
1  class Person:
2      def __init__(self, age):
3          self._age = age
4
5      @property
6      def age(self):
7          return self._age
8
9
10 p1 = Person(18)
11 print(p1.age)
12 p1.age = 19 # Вызовет AttributeError, т.к. сеттер не определен
```

## Getter (Получение значения)

Декоратор `@property` над методом превращает его в атрибут, доступный только для чтения.

```
1  from datetime import date
2  class Person:
3      def __init__(self, birthday: date):
4          self._birthday = birthday
5
6      @property
7      def age(self):
8          today = date.today()
9          age = (today.year - self._birthday.year
10                 - ((today.month, today.day)
11                    < (self._birthday.month, self._birthday.day)
12                     )
13                 )
14      return age
```



## Setter (Установка значения)

Декоратор `@имя_свойства.setter` позволяет определить метод, который будет вызываться при попытке присвоить значение. Здесь и происходит **валидация**.

```
1 class Person:
2     def __init__(self, age):
3         self._age = age
4
5     @property
6     def age(self):
7         return self._age
8
9     @age.setter
10    def age(self, value):
11        if not isinstance(value, int):
12            raise TypeError()
```



## Setter (Установка значения)

Декоратор `@имя_свойства.setter` позволяет определить метод, который будет вызываться при попытке присвоить значение. Здесь и происходит **валидация**.

```
1 class Person:
2     def __init__(self, age):
3         self._age = age
4
5     @property
6     def age(self):
7         return self._age
8
9     @age.setter
10    def age(self, value):
11        if not isinstance(value, int):
12            raise TypeError()
13        if not 0 <= value <= 150:
```



## Setter (Установка значения)

Декоратор `@имя_свойства.setter` позволяет определить метод, который будет вызываться при попытке присвоить значение. Здесь и происходит **валидация**.

```
7
8
9     @age.setter
10    def age(self, value):
11        if not isinstance(value, int):
12            raise TypeError()
13        if not 0 <= value < 150:
14            raise ValueError()
15
16        self._age = value
17
18 p1 = Person(25)
19 print(p1.age) # 25
```



## Setter (Установка значения)

Декоратор `@имя_свойства.setter` позволяет определить метод, который будет вызываться при попытке присвоить значение. Здесь и происходит **валидация**.

```
~      ~~~~~_~~~~ ~~~~  
4  
5     @property  
6     def age(self):  
7         return self._age  
8  
9     @age.setter  
10    def age(self, value):  
11        if not isinstance(value, int):  
12            raise TypeError()  
13        if not 0 <= value < 150:  
14            raise ValueError()  
15
```

## Setter (Установка значения)

Декоратор `@имя_свойства.setter` позволяет определить метод, который будет вызываться при попытке присвоить значение. Здесь и происходит **валидация**.

```
7
8
9     @age.setter
10    def age(self, value):
11        if not isinstance(value, int):
12            raise TypeError()
13        if not 0 <= value < 150:
14            raise ValueError()
15
16        self._age = value
17
18 p1 = Person(25)
19 print(p1.age) # 25
```

## Setter (Установка значения)

Декоратор `@имя_свойства.setter` позволяет определить метод, который будет вызываться при попытке присвоить значение. Здесь и происходит **валидация**.

```
10     def age(self, value):
11         if not isinstance(value, int):
12             raise TypeError()
13         if not 0 <= value < 150:
14             raise ValueError()
15
16         self._age = value
17
18 p1 = Person(25)
19 print(p1.age)      # 25
20 p1.age = 30         # OK, вызовется сеттер
21 p1.age = -5        # Вызовет ValueError
```



```
1 class Person:
2     def __init__(self, age):
3         self._age = age
4
5     def get_age(self):
6         return self._age
7
8     def set_age(self, value):
9         if not isinstance(value, int):
10             raise TypeError()
11         if not 0 <= value < 150:
12             raise ValueError()
13
14         self._age = value
15
```



4

```
5     def get_age(self):
```

```
6         return self._age
```

7

```
8     def set_age(self, value):
```

```
9         if not isinstance(value, int):
```

```
10             raise TypeError()
```

```
11         if not 0 <= value < 150:
```

```
12             raise ValueError()
```

13

```
14         self._age = value
```

15

```
16     def del_age(self):
```

```
17         del self._age
```

18



```
5     def get_age(self):
6         return self._age
7
8     def set_age(self, value):
9         if not isinstance(value, int):
10             raise TypeError()
11         if not 0 <= value < 150:
12             raise ValueError()
13
14         self._age = value
15
16     def del_age(self):
17         del self._age
18
19     age = property(get_age, set_age, del_age)
```



# Функция property

```
1 class Person:
2     def __init__(self, age):
3         self._age = age
4
5
6     def get_age(self):
7         return self._age
8
9     def set_age(self, value):
10        if not isinstance(age, int):
11            raise TypeError()
12        if not 0 <= age < 150:
13            raise ValueError()
14
15        self._age = value
```



# Функция property

```
1 class Person:
2     def __init__(self, age):
3         self._age = age
4
5
6     def get_age(self):
7         return self._age
8
9     def set_age(self, value):
10        if not isinstance(age, int):
11            raise TypeError()
12        if not 0 <= age < 150:
13            raise ValueError()
14
15        self._age = value
```

**Наследование** - способ создания нового класса на основе уже существующего, при котором класс-потомок заимствует свойства и методы родительского класса и также добавляет собственные.


**Наследование** - способ создания нового класса на основе уже существующего, при котором класс-потомок заимствует свойства и методы родительского класса и также добавляет собственные.

1. Класс-потомок = Свойства и методы родителя + Собственные свойства и методы.
2. Класс-потомок автоматически наследует от родительского класса все поля и методы.
3. Класс-потомок может дополняться новыми свойствами.
4. Класс-потомок может дополняться новыми методами, а также заменять(переопределять) унаследованные методы.

**Наследование** - способ создания нового класса на основе уже существующего, при котором класс-потомок заимствует свойства и методы родительского класса и также добавляет собственные.

1. Класс-потомок = Свойства и методы родителя + Собственные свойства и методы.
2. Класс-потомок автоматически наследует от родительского класса все поля и методы.
3. Класс-потомок может дополняться новыми свойствами.
4. Класс-потомок может дополняться новыми методами, а также заменять(переопределять) унаследованные методы.

**Отношение "is-a" (является):** Потомок является разновидностью родителя. Например, Собака является Животным. Кнопка является Элементом Интерфейса. Если это отношение не выполняется, наследование, скорее всего, выбрано неверно. ::

 Дом
Тип фундамента Материал кровли Кол-во окон Кол-во дверей
Построить() Отремонтировать() Заселить() Снести()



## С Дом

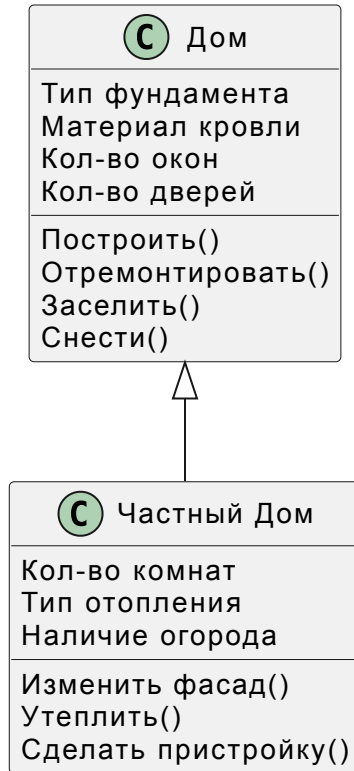
Тип фундамента  
Материал кровли  
Кол-во окон  
Кол-во дверей

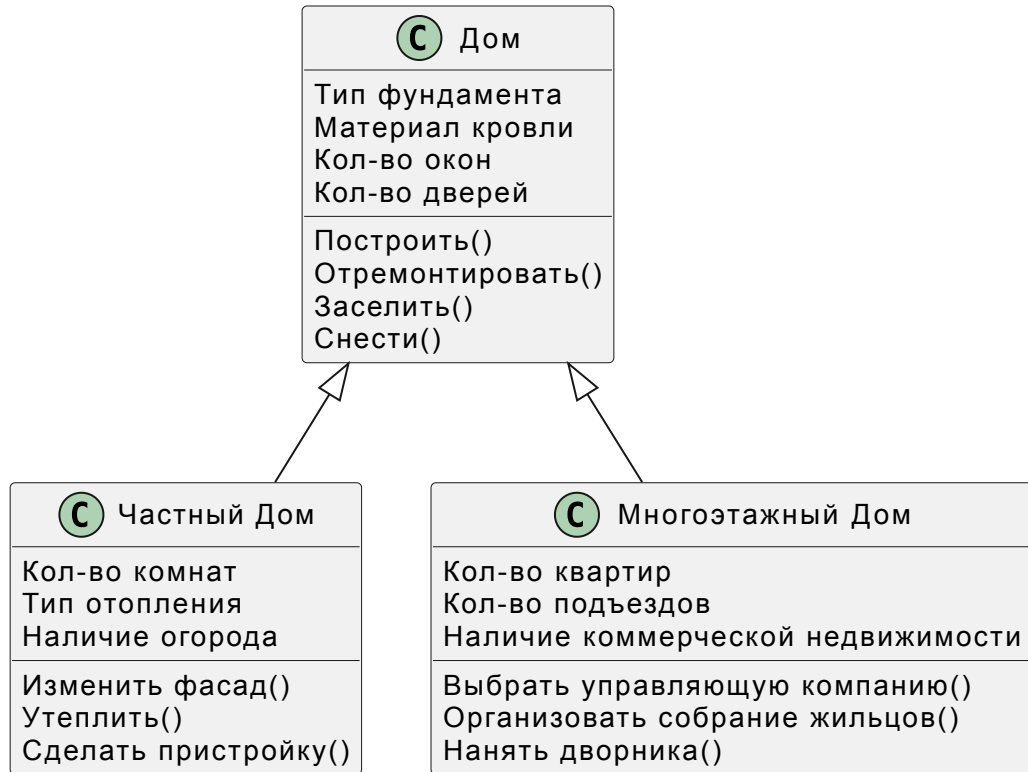
Построить()  
Отремонтировать()  
Заселить()  
Снести()

## С Частный Дом

Тип фундамента  
Материал кровли  
Кол-во окон  
Кол-во дверей  
Кол-во комнат  
Тип отопления  
Наличие огорода

Построить()  
Отремонтировать()  
Заселить()  
Снести()  
Изменить фасад()  
Утеплить()  
Сделать пристройку()





```
1 class Parent:
2     # ...
3
4 class Child(Parent):
5     # ...
```

```
2     def __init__(self, name):
3         self.name = name
4
5     def speak(self):
6         raise NotImplementedError("Потомок должен реализовать этот метод")
7
8 class Dog(Animal):
9     def speak(self): # Переопределение метода родителя
10         return f"{self.name} говорит Гав!"
11
12 class Cat(Animal):
13     def purr(self): # Расширение - новый метод
14         return f"{self.name} мурлычет"
15
16     def speak(self): # Переопределение метода родителя
```

Встроенная функция `super()` предоставляет доступ к методам родительского класса.

```
1 class Employee:
2     def __init__(self, name, salary):
3         self.name = name
4         self.salary = salary
5
6     def display_info(self):
7         print(f"Имя: {self.name}, Зарплата: {self.salary}")
8
9 class Manager(Employee):
10    def __init__(self, name, salary, department):
11        # Сначала вызываем __init__ родителя, чтобы он сделал свою работу
12        super().__init__(name, salary)
13        # Затем добавляем свою логику
```

Встроенная функция `super()` предоставляет доступ к методам родительского класса.

```
6     def display_info(self):
7         print(f"Имя: {self.name}, Зарплата: {self.salary}")
8
9     class Manager(Employee):
10        def __init__(self, name, salary, department):
11            # Сначала вызываем __init__ родителя, чтобы он сделал свою работу
12            super().__init__(name, salary)
13            # Затем добавляем свою логику
14            self.department = department
15
16        def display_info(self):
17            # Сначала вызываем метод родителя
18            super().display_info()
```

Встроенная функция `super()` предоставляет доступ к методам родительского класса.

```
6     def display_info(self):
7         print(f"Имя: {self.name}, Зарплата: {self.salary}")
8
9     class Manager(Employee):
10        def __init__(self, name, salary, department):
11            # Сначала вызываем __init__ родителя, чтобы он сделал свою работу
12            super().__init__(name, salary)
13            # Затем добавляем свою логику
14            self.department = department
15
16        def display_info(self):
17            # Сначала вызываем метод родителя
18            super().display_info()
```

Встроенная функция `super()` предоставляет доступ к методам родительского класса.

```
1 class Employee:
2     def __init__(self, name, salary):
3         self.name = name
4         self.salary = salary
5
6     def display_info(self):
7         print(f"Имя: {self.name}, Зарплата: {self.salary}")
8
9 class Manager(Employee):
10    def __init__(self, name, salary, department):
11        # Сначала вызываем __init__ родителя, чтобы он сделал свою работу
12        super().__init__(name, salary)
13        # Затем добавляем свою логику
```

Встроенная функция `super()` предоставляет доступ к методам родительского класса.

```
8
9 class Manager(Employee):
10     def __init__(self, name, salary, department):
11         # Сначала вызываем __init__ родителя, чтобы он сделал свою работу
12         super().__init__(name, salary)
13         # Затем добавляем свою логику
14         self.department = department
15
16     def display_info(self):
17         # Сначала вызываем метод родителя
18         super().display_info()
19         # Затем дополняем его вывод
20         print(f"Отдел: {self.department}")
```

Встроенная функция `super()` предоставляет доступ к методам родительского класса.

```
8
9 class Manager(Employee):
10     def __init__(self, name, salary, department):
11         # Сначала вызываем __init__ родителя, чтобы он сделал свою работу
12         super().__init__(name, salary)
13         # Затем добавляем свою логику
14         self.department = department
15
16     def display_info(self):
17         # Сначала вызываем метод родителя
18         super().display_info()
19         # Затем дополняем его вывод
20         print(f"Отдел: {self.department}")
```

Встроенная функция `super()` предоставляет доступ к методам родительского класса.

```
1 class Employee:
2     def __init__(self, name, salary):
3         self.name = name
4         self.salary = salary
5
6     def display_info(self):
7         print(f"Имя: {self.name}, Зарплата: {self.salary}")
8
9 class Manager(Employee):
10    def __init__(self, name, salary, department):
11        # Сначала вызываем __init__ родителя, чтобы он сделал свою работу
12        super().__init__(name, salary)
13        # Затем добавляем свою логику
```

```
1 class Child(Parent1, Parent2):  
2     # ...
```

```
1 class A:
2     def info(self): print("In class A")
3
4 class B(A):
5     def info(self): print("In class B")
6
7 class C(A):
8     def info(self): print("In class C")
9
10 class D(B, C):
11     pass
12
13 d = D()
14 d.info()
```

```
1 class A:
2     def info(self): print("In class A")
3
4 class B(A):
5     def info(self): print("In class B")
6
7 class C(A):
8     def info(self): print("In class C")
9
10 class D(B, C):
11     pass
12
13 d = D()
14 d.info() # Выведет "In class B"
```

```
1 class A:
2     def info(self): print("In class A")
3
4 class B(A):
5     def info(self): print("In class B")
6
7 class C(A):
8     def info(self): print("In class C")
9
10 class D(B, C):
11     pass
12
13 d = D()
14 d.info() # Выведет "In class B"
15 print(D.__mro__)
16 # Выведет: (<class '__main__.D'>, <class '__main__.B'>, <class '__main__
```

Множественное наследование — мощный, но опасный инструмент. Часто его **можно** и **нужно** заменять композицией или миксинами.