

Лекционный материал: Наследование и полиморфизм в Python

◆ Что такое наследование?

Наследование — это механизм ООП, позволяющий одному классу (дочернему) унаследовать атрибуты и методы другого класса (родительского). Это способ **повторного использования кода** и создания иерархий.

```
class Parent:
    def greet(self):
        print("Привет из родителя!")

class Child(Parent): # наследование
    pass

obj = Child()
obj.greet() # Привет из родителя!
```

✓ ☒ **Дочерний класс может расширять** функциональность (добавлять новые методы) и **переопределять** существующие.

◆ Переопределение методов

Дочерний класс может задать **свою реализацию** метода с тем же именем:

```
class Animal:
    def speak(self):
        print("Животное издаёт звук")

class Dog(Animal):
    def speak(self): # переопределение
        print("Гав-гав!")
```

◆ Что такое полиморфизм?

Полиморфизм (от греч. «много форм») — это возможность **работать с объектами разных типов единообразно**, если у них есть общий интерфейс (например, одинаковые методы).

```
animals = [Dog(), Cat(), Bird()]
```

```
for animal in animals:
    animal.speak() # каждый издаёт свой звук
```

- ✓ Один вызов — разное поведение в зависимости от типа объекта.
- ✓ Это делает код **гибким, расширяемым и легко тестируемым**.

◆ Абстрактные классы и `@abstractmethod` (опционально)

Чтобы гарантировать, что все потомки реализуют определённый метод, можно использовать **абстрактный базовый класс (ABC)**:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Гав!"
```

Если не реализовать `speak()` в `Dog`, при создании объекта будет ошибка.

✿ Сложная задача: Система оплаты — Полиморфизм в действии

📌 Условие задачи

Вы разрабатываете систему онлайн-магазина. В ней должна поддерживаться **разная логика расчёта итоговой стоимости** в зависимости от типа клиента:

- **Обычные клиенты** платят полную цену.
- **Премиум-клиенты** получают **10% скидку** на всё.
- **Корпоративные клиенты** платят **полную цену**, но им **начисляются бонусные баллы** (1 балл = 1 рубль покупки).

Кроме того, все клиенты могут применять **промокоды**, которые дают **фиксированную скидку в рублях** (например, «SAVE50» = −50 Р), но **не могут сделать итоговую сумму отрицательной**.

Ваша задача — спроектировать иерархию классов так, чтобы:

1. Был общий интерфейс для расчёта итоговой стоимости.
2. Каждый тип клиента реализовывал свою логику.
3. Промокод применялся **после** расчёта базовой стоимости (со скидками или без).

4. Для корпоративных клиентов отдельно хранились и обновлялись бонусные баллы.

Требования к реализации

Создайте следующие классы:

1. Базовый класс `Customer`

- Атрибуты:
 - `name` (str)
 - `total_spent` (float) — общая сумма всех покупок (для статистики)
- Методы:
 - `calculate_discounted_price(base_price)` — **абстрактный метод**, должен быть реализован в потомках.
 - `apply_promo(final_price, promo_discount)` — принимает итоговую цену и скидку от промокода, возвращает `max(0, final_price - promo_discount)`.
 - `make_purchase(base_price, promo_discount=0)` — основной метод покупки:
 - Вызывает `calculate_discounted_price(base_price)`
 - Применяет промокод через `apply_promo()`
 - Обновляет `total_spent`
 - Возвращает итоговую сумму к оплате

2. Дочерние классы

`RegularCustomer(Customer)`

- Без скидок: `calculate_discounted_price` возвращает `base_price`.

`PremiumCustomer(Customer)`

- Скидка 10%: возвращает `base_price * 0.9`.

`CorporateCustomer(Customer)`

- Атрибут: `bonus_points` (int, изначально 0)
 - `calculate_discounted_price` возвращает `base_price` (без скидки).
 - В методе `make_purchase` после расчёта итоговой суммы **добавляет к `bonus_points` значение `base_price`** (до промокода!).
-

Пример использования

```
# Создаём клиентов
regular = RegularCustomer("Иван")
premium = PremiumCustomer("Анна")
corp = CorporateCustomer("000 'Ромашка'")

# Покупки
```

```
price = 1000

# Обычный клиент с промокодом на 100 руб
print(regular.make_purchase(price, promo_discount=100)) # 900.0

# Премиум-клиент с тем же промокодом
print(premium.make_purchase(price, promo_discount=100)) # 800.0 (1000*0.9 - 100)

# Корпоративный клиент
print(corp.make_purchase(price, promo_discount=200)) # 800.0
print(corp.bonus_points) # 1000 (базовая цена, не итоговая!)
print(corp.total_spent) # 800.0
```

✓ Подробное решение

Шаг 1: Импортируем ABC и abstractmethod

```
from abc import ABC, abstractmethod
```

Шаг 2: Базовый класс Customer

```
class Customer(ABC):
    def __init__(self, name):
        self.name = name
        self.total_spent = 0.0

    @abstractmethod
    def calculate_discounted_price(self, base_price):
        """Рассчитывает цену после скидок клиента (до промокода)."""
        pass

    def apply_promo(self, final_price, promo_discount):
        """Применяет промокод и не даёт уйти в минус."""
        return max(0.0, final_price - promo_discount)

    def make_purchase(self, base_price, promo_discount=0):
        """Основной метод покупки."""
        discounted = self.calculate_discounted_price(base_price)
        final_price = self.apply_promo(discounted, promo_discount)
        self.total_spent += final_price
        return final_price
```

🔍 Обратите внимание:

- `make_purchase` — общий для всех.

- Конкретная логика скидок — в `calculate_discounted_price`, который каждый потомок реализует по-своему.

Шаг 3: Класс `RegularCustomer`

```
class RegularCustomer(Customer):  
    def calculate_discounted_price(self, base_price):  
        return base_price # без скидок
```

Шаг 4: Класс `PremiumCustomer`

```
class PremiumCustomer(Customer):  
    def calculate_discounted_price(self, base_price):  
        return base_price * 0.9 # 10% скидка
```

Шаг 5: Класс `CorporateCustomer`

```
class CorporateCustomer(Customer):  
    def __init__(self, name):  
        super().__init__(name)  
        self.bonus_points = 0  
  
    def calculate_discounted_price(self, base_price):  
        return base_price # без скидки  
  
    def make_purchase(self, base_price, promo_discount=0):  
        # Сначала начисляем бонусы по БАЗОВОЙ цене  
        self.bonus_points += int(base_price)  
        # Затем вызываем родительскую логику расчёта  
        return super().make_purchase(base_price, promo_discount)
```

💡 Здесь мы **расширяем** поведение `make_purchase`, а не заменяем его полностью.
Сначала — бонусы, потом — стандартный расчёт через `super()`.

Проверка работы

```
# Тест из условия  
regular = RegularCustomer("Иван")  
premium = PremiumCustomer("Анна")  
corp = CorporateCustomer("ООО 'Ромашка'")  
  
price = 1000
```

```
print(regular.make_purchase(price, 100)) # 900.0
print(premium.make_purchase(price, 100)) # 800.0
print(corp.make_purchase(price, 200))    # 800.0
print("Бонусы:", corp.bonus_points)     # 1000
print("Потрачено:", corp.total_spent)    # 800.0
```

🤖 Почему это демонстрирует полиморфизм?

- Все клиенты имеют **единый интерфейс**: метод `make_purchase`.
- Мы можем обрабатывать их **одинаково**, даже не зная их точного типа:

```
clients = [regular, premium, corp]
for client in clients:
    cost = client.make_purchase(500, 50)
    print(f"{client.name} заплатил {cost}")
```

📌 Ключевые выводы

1. **Наследование** позволило вынести общую логику (`apply_promo`, `total_spent`) в базовый класс.
2. **Абстрактный метод** гарантировал, что каждый тип клиента реализует свою скидку.
3. **Полиморфизм** позволяет работать со всеми клиентами через один интерфейс.
4. **Расширение поведения** (в `CorporateCustomer`) показывает гибкость ООП.

🩺 Практическая задача (60 минут): Ветеринарная клиника — Приём разных животных

📌 Сценарий

В ветеринарной клинике работают с **разными видами животных**:

- **Собаки** — нуждаются в прививке от бешенства и любят играть.
- **Кошки** — требуют обработки от блох и часто мурлычут.
- **Птицы** — нуждаются в проверке перьев и издаю звуки.

Каждое животное приходит на приём, и ветеринар **выполняет стандартную процедуру**, но **для каждого вида — свои действия**.

Ваша задача — создать иерархию классов, где все животные имеют **общий интерфейс приёма**, но **выполняют разные действия**.

🎯 Требования к реализации

1. Базовый класс `Animal`

- **Атрибуты** (все публичные):
 - `name` — кличка (строка)
 - `species` — вид (строка, например, "Собака")
 - **Методы:**
 - `__init__(self, name)`
 - `examine()` — **должен быть переопределён** в дочерних классах.
В базовом классе: `print(f"{self.name} осмотрен.")`
 - `__str__()` — возвращает: `"{name} ({species})"`
-

2. Дочерние классы

a) `Dog(Animal)`

- В `__init__` принимает `name`.
- `species = "Собака"`
- Метод `examine()`:

```
print(f"{self.name} получил прививку от бешенства.")
print(f"{self.name} поиграл с мячиком.")
```

b) `Cat(Animal)`

- В `__init__` принимает `name`.
- `species = "Кошка"`
- Метод `examine()`:

```
print(f"{self.name} обработан от блох.")
print(f"{self.name} громко мурлычет.")
```

c) `Bird(Animal)`

- В `__init__` принимает `name`.
- `species = "Птица"`
- Метод `examine()`:

```
print(f"Перья {self.name} проверены.")
print(f"{self.name} чирикает радостно.")
```

💡 Во всех дочерних классах вызывайте `super().__init__(name)` и **устанавливайте** `self.species` вручную.

3. Функция приёма в клинике

Напишите функцию:

```
def vet_appointment(animals):  
    """  
    Принимает список животных.  
    Для каждого:  
    - выводит str(животное)  
    - вызывает examine()  
    - выводит пустую строку для разделения  
    """
```

Эта функция **не знает**, какие именно животные ей передали — она просто вызывает единый метод `examine()`.

4. Демонстрация

Создайте список из 5 животных:

```
patients = [  
    Dog("Бобик"),  
    Cat("Мурка"),  
    Bird("Чирик"),  
    Dog("Рекс"),  
    Cat("Барсик")  
]
```

Вызовите `vet_appointment(patients)`.



Пример ожидаемого вывода

```
Бобик (Собака)  
Бобик получил прививку от бешенства.  
Бобик поиграл с мячиком.  
  
Мурка (Кошка)  
Мурка обработан от блох.  
Мурка громко мурлычет.  
  
Чирик (Птица)  
Перья Чирик проверены.  
Чирик чирикает радостно.  
  
Рекс (Собака)  
Рекс получил прививку от бешенства.
```


Рекс поиграл с мячиком.

Барсик (Кошка)

Барсик обработан от блох.


Барсик громко мурлычет.

Подсказки для студентов

- В дочерних классах после `super().__init__(name)` напишите:
`self.species = "Собака"` (и т.д.)
- В функции `vet_appointment` просто вызывайте `animal.examine()` — полиморфизм сделает всё сам.
- Не нужно использовать `isinstance()` — это **не требуется**, так как интерфейс общий.

Почему это наследование и полиморфизм?

- **Наследование:** все животные наследуют `name` и метод `__str__`.
- **Полиморфизм:** вызов `examine()` выглядит одинаково, но:
 - У собаки — прививка и игра,
 - У кошки — обработка и мурлыканье,
 - У птицы — проверка перьев и чириканье.

 Это **чистый полиморфизм**: один интерфейс — много реализаций.

Что сдать

- Один файл `.py` с классами `Animal`, `Dog`, `Cat`, `Bird`.
 - Функция `vet_appointment`.
 - Демонстрационный код с 5 животными.
 - Без `input()`, без `_`, без `@property`, без валидации.
-