IT CAREER HUB

Урок 36.
Наследование и полиморфизм

IT CAREER HUB

Принципы ООП

Объектно-ориентированное программирование основывается на **четырёх ключевых принципах**, которые помогают **структурировать код, упростить его поддержку и расширение**.

Эти принципы работают вместе и делают программы **гибкими, модульными и удобными в расширении**.

Принцип	**Краткое описание**
Наследование	Один класс может **унаследовать** свойства и поведение другого. Это позволяет **избегать дублирования кода** и **переиспользовать логику**.
Инкапсуляция	Объект **скрывает внутреннее устройство** и предоставляет **только нужный интерфейс**. Позволяет защищать данные и контролировать доступ к ним.
Полиморфизм	Одинаковый интерфейс может **вести себя по-разному** в зависимости от типа объекта. Позволяет использовать **одинаковый код для разных классов**.
Абстракция	Выделение **общих характеристик и поведения**, чтобы задать интерфейс для будущих реализаций в наследниках.

Зачем знать принципы?

- Помогают **понять, как правильно проектировать классы и объекты**
- Упрощают **понимание механизмов наследования, переопределения и взаимодействия объектов**
- Позволяют **писать гибкий, расширяемый и поддерживаемый код**

IT CAREER HUB

Наследование

Наследование — это механизм, который позволяет одному классу перенять свойства и поведение другого. Это один из ключевых принципов ООП, который помогает избегать дублирования кода и повторно использовать общую логику.

Кто кого наследует?

- Класс, от которого наследуются — называется **родительский класс**, **базовый класс** или **суперкласс**.
- Класс, который наследует — называется **дочерний класс**, **производный класс** или **подкласс**.

Зачем нужно наследование?

- Чтобы **избежать дублирования кода**, выделив общее поведение в один базовый класс
- Чтобы **удобно расширять** существующую логику без переписывания старых классов
- Чтобы **работать с разными объектами одинаково**, если они наследуются от общего предка

Синтаксис:

```python
class Parent:
    # родительский класс
    ...

class Child(Parent):
    # дочерний класс, наследует всё от Parent
    ...
```

IT CAREER HUB

Пример: наследование полей и методов

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

    def work(self):
        print(f"{self.name} is working...")

class Programmer(Employee):
    pass

class Manager(Employee):
    pass

programmer = Programmer("Alice")
manager = Manager("Bob")

programmer.work()
manager.work()
```

- `Employee` — базовый класс, который хранит имя сотрудника и умеет работать.
- `Programmer` и `Manager` **наследуют всё от** `Employee`:
  и поле `name`, и метод `work()`.
- Каждый из них может использовать эти возможности **без повторного определения**.

Особенности:

- Дочерний класс **наследует все поля и методы** родительского класса
- Можно **добавлять свои методы и поля**, не затрагивая родителя
- Можно **переопределять поведение**, если нужно изменить работу унаследованного метода

Полиморфизм

Полиморфизм (от греч. «много форм») — это возможность использовать один и тот же интерфейс для разных типов объектов.
Иными словами, **разные классы могут реализовывать один и тот же метод по-своему**, а вызывающий код будет работать с ними **одинаково**.

Зачем нужен полиморфизм?

- Позволяет **писать универсальный код**, не заботясь о типе объекта
- Упрощает **расширение** программы — можно добавлять новые типы, не меняя старую логику
- Делает код **гибким и читаемым**

Механизмы полиморфизма

Полиморфизм реализуется через два подхода:

1. Переопределение методов (overriding):

Когда **дочерний класс заново определяет метод**, унаследованный от родителя.
При вызове такого метода используется **реализация из дочернего класса**.

2. Перегрузка (overloading):

Когда **один и тот же метод** может вести себя по-разному **в зависимости от переданных аргументов**.
В Python нет классической перегрузки, как в других языках (Java, C++).

Пример: переопределение метода

```python
class Employee:

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

```
    def work(self):
        print(f"{self.name} is working...")

class Programmer(Employee):
    def work(self):
        print(f"{self.name} writing code...")

class Manager(Employee):
    def work(self):
        print(f"{self.name} managing team...")


staff = [Programmer("Alice"), Manager("Bob"), Programmer("Bill")]

for person in staff:
    person.work()  # Поведение зависит от конкретного типа
```

- У всех сотрудников есть метод `work()`, но он реализован по-разному.
- В списке `staff` мы храним объекты разных классов: `Programmer`, `Manager`.
- При вызове `person.work()` Python **автоматически вызывает нужную версию метода** в зависимости от того, к какому классу относится объект.
- Это и есть **полиморфизм**: **один вызов → разное поведение в зависимости от типа объекта**.

Пример: попытка перегрузки (не сработает)

```
Python
class Math:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

m = Math()
print(m.add(1, 2, 3))   # Работает
```

```
print(m.add(1, 2))      # Ошибка!
```

- Python не поддерживает **перегрузку** методов по количеству аргументов.
- В примере выше вторая версия `add()` просто **заменяет** первую — Python запоминает только последнее определение.

IT CAREER HUB

Задания для закрепления 1

1. Что произойдёт при выполнении следующего кода?

```python
class Employee:
    def work(self):
        print("Employee is working")

class Manager(Employee):
    pass

person = Manager()
person.work()
```

a. Возникнет ошибка, так как `Manager` не имеет метода `work()`
b. Будет напечатано: `Employee is working`
c. Будет напечатано: `None`
d. Метод `work()` выполнится, но ничего не выведет

Посмотреть ответ

2. Укажи, что относится к преимуществам наследования:

a. Повышение безопасности данных
b. Возможность писать меньше кода
c. Возможность скрыть поля от внешнего доступа
d. Возможность переиспользовать логику

Посмотреть ответ

IT CAREER HUB

Функция super

При наследовании часто возникает ситуация, когда родительский класс уже задаёт общие поля в `__init__()`, а в дочернем классе необходимо добавить новые поля.

Чтобы **не дублировать одинаковый код** используется функция `super()` — она вызывает ближайший метод родителя.

Пример: неправильный способ - дублируем код родителя

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

class Programmer(Employee):
    def __init__(self, name, language):
        self.name = name  # повторяем то же, что уже делает родитель
        self.language = language

class Manager(Employee):
    def __init__(self, name, department):
        self.name = name  # снова повторяем
        self.department = department
```

Такой код **нарушает DRY** (Don't Repeat Yourself) и делает поддержку сложнее: если поведение `Employee.__init__` поменяется — `Programmer` об этом не узнает.

Пример: альтернативный неправильный способ - вызываем код родителя

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

```
class Programmer(Employee):
    def __init__(self, name, language):
        Employee.__init__(self, name)  # явный вызов родителя
        self.language = language
```

Этот способ работает, но считается **плохой практикой**:

- Привязан к имени родительского класса
- Может некорректно работать при изменении иерархии
- Не учитывает порядок разрешения методов

Пример: правильный способ - использование super()

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

class Programmer(Employee):
    def __init__(self, name, language):
        super().__init__(name)  # вызываем родительский __init__
        self.language = language

class Manager(Employee):
    def __init__(self, name, department):
        super().__init__(name)  # вызываем родительский __init__
        self.department = department

p = Programmer("Alice", "Python")
print(p.name)
print(p.language)
```

Функция super в цепочке наследования

Когда у нас есть **несколько уровней наследования**, `super` позволяет вызывать инициализацию каждого уровня **без жёсткой привязки к имени родительского класса**.

Пример: многоуровневое наследование

```python
class Person:
    def __init__(self, name):
        self.name = name
        print(f"Init Person: {self.name}")

class Employee(Person):
    def work(self):
        print(f"{self.name} is working...")


class Manager(Employee):
    def __init__(self, name, department):
        super().__init__(name)
        self.department = department
        print(f"Init Manager: {self.name} manages {self.department}")

m = Manager("Alice", "Development")
```

Функция super в обычных методах

Функция `super` полезна не только в `__init__()`, но и **в любых других методах**, где нужно **расширить поведение родителя**, а не полностью его заменить.

Пример: расширение обычного метода

```python
class Employee:
    def work(self):
        print("Employee is doing general tasks.")

class Programmer(Employee):
    def work(self):
```

```
        super().work()  # вызываем метод родителя
        print("Programmer is writing code.")

class Manager(Employee):
    def work(self):
        super().work()
        print("Manager is holding a meeting.")

staff = [Programmer(), Manager()]
for person in staff:
    person.work()
    print()
```

Чем эффективен super?

- Автоматически находит метод в ближайшем родителе
- Позволяет избежать дублирования кода
- Гарантирует корректную работу при изменении иерархии классов

Когда использовать super?

- Когда **родитель уже делает нужную инициализацию**, и мы хотим её сохранить
- Когда **расширяем** (а не полностью заменяем) поведение метода
- Особенно важно при **множественном наследовании** — `super()` автоматически учитывает порядок вызова

IT CAREER HUB

Задания для закрепления 2

1. **Найди ошибку в коде:**

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

class Programmer(Employee):
    def __init__(self, name, language):
        super(name).__init__()
        self.language = language
```

Посмотреть ответ

IT CAREER HUB

Наследование от object

В Python все классы **неявно наследуются** от встроенного класса `object`, даже если это явно не указывается.

Класс `object` — это корневой класс всей иерархии классов в Python. Он предоставляет набор базовых методов, таких как `__str__()`, `__init__()` и другие.

Пример:

```python
class Book:
    pass
```

Даже если мы не указываем родителя, Python воспринимает это так:

```python
class Book(object):
    pass
```

Методы от object

Метод `__str__()` **присутствует в классе без собственной реализации**, поскольку он унаследован от базового класса `object`.

```python
class Book:
    pass

b = Book()

print(b.__str__())  # Вызов метода __str__ напрямую
print(b)            # Вызов метода __str__
```

Преимущества наследования от object

- Все классы получают **единое поведение по умолчанию**
- Все объекты можно безопасно использовать с базовыми функциями и методами
- Это делает классы совместимыми со встроенными механизмами Python (например, `print()` вызывает `__str__()` из `object`, если не переопределено)

Функции isinstance и issubclass

Python предоставляет удобные встроенные функции для **проверки типов и иерархии классов**, что особенно полезно при работе с **наследованием** и **полиморфизмом**.

Функция isinstance

isinstance — это встроенная функция Python, которая позволяет проверить, является ли объект экземпляром определённого класса или его подкласса, чтобы выбрать правильное поведение

Если объект принадлежит **указанному классу или его наследнику**, функция возвращает `True`, в противном случае — `False`.

Функция `isinstance` также работает со **встроенными типами** (`str`, `list`, `int` и т.д.)

Синтаксис:

```python
isinstance(obj, some_class)
isinstance(obj, tuple_of_classes)
```

- `obj` — объект, который необходимо проверить
- `some_class` — класс, принадлежность к которому необходимо проверить
- `tuple_of_classes` — кортеж из нескольких классов, принадлежность к любому из которых необходимо проверить

Пример

```python
class Employee:
    pass
```

```
class Programmer(Employee):
    pass

class Manager(Employee):
    pass

e = Employee()
p = Programmer()
m = Manager()

print(isinstance(p, Programmer))  # экземпляр класса
print(isinstance(p, Employee))    # экземпляр наследника
print(isinstance(p, Manager))     # Manager не находится выше в иерархии
print(isinstance(p, object))      # True – все классы наследуют от object
```

Пример использования в коде

Допустим, у некоторых сотрудников есть метод `write_code()`, но не у всех. Чтобы не получить ошибку при вызове, проверим, принадлежит ли объект к нужному классу.

Python

```
class Employee:
    def work(self):
        print("Выполняет общие задачи")

class Programmer(Employee):
    def write_code(self):
        print("Пишет код")

class BackendDeveloper(Programmer):
    def write_code(self):
        print("Пишет серверный код")

class FrontendDeveloper(Programmer):
    def write_code(self):
        print("Пишет интерфейс")
```

```
class Manager(Employee):
    def work(self):
        print("Проводит собрание")


staff = [
    Programmer(),
    BackendDeveloper(),
    FrontendDeveloper(),
    Manager(),
    Employee()
]

for person in staff:
    if isinstance(person, Programmer):
        person.write_code()
    else:
        person.work()
```

Проверка на несколько классов

Можно проверить, принадлежит ли объект **к одному из нескольких классов**, передав кортеж:

```
Python
print(isinstance("hello", (str, int)))  # Строка принадлежит к одному из
классов
```

Функция issubclass

issubclass — это встроенная функция Python, которая позволяет проверить, является ли один класс подклассом другого, то есть наследуется ли он от указанного класса.

Функция возвращает `True`, если класс **наследуется от другого класса напрямую или через цепочку**, и `False` — если нет.

Синтаксис:

```python
issubclass(class_a, class_b)
issubclass(class_a, tuple_of_classes)
```

- `class_a` — класс, который мы проверяем
- `class_b` — предполагаемый родительский класс
- `tuple_of_classes` — кортеж из нескольких классов

Пример

```python
class Employee:
    pass

class Programmer(Employee):
    pass

class Manager(Employee):
    pass

class BackendDeveloper(Programmer):
    pass

print(issubclass(Programmer, Employee))         # Прямой потомок
print(issubclass(BackendDeveloper, Programmer)) # Прямой потомок
print(issubclass(BackendDeveloper, Employee))   # Потомок через цепочку
print(issubclass(Manager, Programmer))          # Разные ветки иерархии
print(issubclass(Employee, object))             # Все классы наследуют от
object
```

Пример использования в коде

Допустим, мы создаём сотрудников и сразу хотим **отправить им приветственное письмо**, с разным содержанием в зависимости от роли. Проверим, что переданный класс — это **подкласс** `Employee`.

Python

```python
class Employee:
    def send_welcome_email(self):
        print("Добро пожаловать в компанию!")

class Programmer(Employee):
    def send_welcome_email(self):
        print("Добро пожаловать, разработчик! Не забудьте подключиться к
репозиторию.")

class Manager(Employee):
    def send_welcome_email(self):
        print("Добро пожаловать, менеджер! Сегодня у вас первое собрание с
командой.")

def hire_employee(cls):
    if not issubclass(cls, Employee):
        raise ValueError("Можно нанимать только классы, основанные на
Employee")

    person = cls()
    person.send_welcome_email()

hire_employee(Programmer)
hire_employee(Manager)
# hire_employee(str)  # ValueError: Не наследник Employee
```

IT CAREER HUB

Задания для закрепления 3

1. **Что вернёт функция issubclass() и почему?**

```python
class Book:
    pass

b = Book()
print(issubclass(b, object))
```

Посмотреть ответ

2. **Для чего используется функция isinstance?**

a. Проверяет, является ли объект экземпляром заданного класса или его наследников
b. Проверяет, что класс унаследован от object
c. Проверяет тип переменной только для встроенных типов

Посмотреть ответ

IT CAREER HUB

Ответы на задания

Задания на закрепление 1	Вернуться к заданиям
1. Результат выполнения кода	Ответ: b
2. Преимущества наследования	Ответ: b, d
Задания на закрепление 2	Вернуться к заданиям
1. Ошибка в коде	Ответ: Неверное использование `super`: должно быть `super().__init__(name)`
Задания на закрепление 3	Вернуться к заданиям
1. Функция `issubclass()`	Ответ: `TypeError`, так как функция `issubclass()` принимает класс, а не объект.
2. Функция `isinstance`	Ответ: a

IT CAREER HUB

Практическая работа

1. Класс Employee

Создайте класс `Employee`, представляющий сотрудника.

- У каждого объекта должно быть поле `name`.
- Метод `work()` выводит строку: `<имя> is working...`.
- Проверьте работу класса, создав сотрудника и вызвав метод `work()`.

Пример вывода:

```python
Alice is working...
```

Решение:

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

    def work(self):
        print(f"{self.name} is working...")

e = Employee("Alice")
e.work()
```

IT CAREER HUB

2. Класс Developer

Создайте класс `Developer`, который расширяет `Employee`.

- Добавьте дополнительное поле `language`.
- Переопределите метод `work()`, чтобы он включал сообщение из родительского метода и добавлял строку:
  `<имя> writes <язык> code.`
- Проверьте работу, создав объект `Developer` и вызвав метод `work()`.

Пример вывода:

```python
Bob is working...
Bob writes Python code.
```

IT CAREER HUB

Решение:

```python
class Developer(Employee):
    def __init__(self, name, language):
        super().__init__(name)
        self.language = language

    def work(self):
        super().work()
        print(f"{self.name} writes {self.language} code.")

d = Developer("Bob", "Python")
d.work()
```