IT CAREER HUB

Урок 38.
Инкапсуляция

IT CAREER HUB

Инкапсуляция

 **Инкапсуляция — это подход, при котором внутреннее устройство объекта скрывается, а взаимодействие с ним осуществляется через понятный и контролируемый интерфейс.**

Это позволяет защищать данные от прямого доступа и управлять тем, **что именно и как можно изменять извне**.

Инкапсуляция помогает:

- **Защитить внутренние данные** от прямого доступа и случайных изменений
- **Контролировать поведение объекта** через ограниченные точки входа
- **Упростить использование объектов** без знания их внутренней структуры

IT CAREER HUB

Уровни доступа

В Python можно разделить атрибуты (переменные и методы) объекта на **три уровня доступа**:

- **Публичные** — доступны отовсюду
- **Защищённые** — предназначены только для использования внутри класса и его наследников
- **Приватные** — скрыты от внешнего кода и не наследуются напрямую

Python не вводит жёстких ограничений доступа, как это делают некоторые другие языки, но предлагает **принятые соглашения по именованию**, которые помогают разработчикам соблюдать инкапсуляцию.

Публичные атрибуты

 **Публичные атрибуты — это атрибуты, к которым можно свободно обращаться из любого места: и изнутри класса, и снаружи — через объект. Они не имеют специального префикса и являются полностью открытыми.**

Особенности:

- Доступны для чтения и записи извне
- Используются как часть **открытого интерфейса** объекта
- Не скрывают детали реализации

 **Пример**

```python
class Book:
    def __init__(self, title, author):
        self.title = title     # публичное поле
        self.author = author   # публичное поле


book = Book("1984", "George Orwell")
print(book.title)           # доступ к публичному полю
```

```
book.title = "Animal Farm"  # изменение публичного поля
print(book.title)
```

- Если какое-то значение должно быть видно и доступно извне, его делают публичным.

Защищённые атрибуты

 **Защищённые атрибуты предназначены для внутреннего использования в классе и его наследниках, но всё ещё могут быть доступны извне.**

В Python защищённые атрибуты обозначаются **одним подчёркиванием** перед именем: _name.

Особенности:

- Условно считаются **не для внешнего использования**
- Не мешают доступу извне, но **сигнализируют**, что это **внутренняя часть реализации**
- Могут использоваться в дочерних классах

 **Пример**

Python
```
class Book:
    def __init__(self, title):
        self._title = title  # защищённый атрибут

    def show_title(self):
        print(self._title)

class SpecialBook(Book):
    def show_title(self):
        print(self._title.upper())  # доступ из наследника

book = SpecialBook("Brave New World")
```

```
print(book._title)        # доступ извне технически возможен
book.show_title()
```

- Python не запрещает использовать `_title` напрямую, но появляется "warning" в виде подчёркивания.

Приватные атрибуты

 **Приватные атрибуты — это атрибуты, которые предназначены только для использования внутри класса. Python не полностью запрещает доступ к приватным атрибутам, но автоматически меняет их имя внутри класса, чтобы затруднить случайное использование извне.**

Приватные атрибуты обозначаются **двумя подчёркиваниями** перед именем: `__name`.

Особенности:

- Нельзя обратиться напрямую извне под тем же именем
- Используются для **скрытия реализации** и защиты важных данных
- Всё ещё технически доступны, но под **изменённым именем**

 **Пример**

```python
class Book:
    def __init__(self, title):
        self.__title = title  # приватный атрибут

    def show_title(self):
        print(self.__title)

class SpecialBook(Book):
    def print_title(self):
        print(self.__title.upper())  # ошибка
```

```
book = SpecialBook("Brave New World")
book.show_title()     # доступ через метод базового класса
# book.shout_title()  # ошибка: такого атрибута с таким именем нет
# book.__title        # ошибка: такого атрибута с таким именем нет
```

Манглирование имён

 **Манглирование имён (от англ. *name mangling*) — это механизм, с помощью которого Python автоматически изменяет имя приватных атрибутов, чтобы они не конфликтовали с атрибутами в дочерних классах и не были случайно переопределены или использованы извне.**

Если атрибут начинается с **двух подчёркиваний** и **не заканчивается** подчёркиваниями: `__value`, Python преобразует имя этого атрибута внутри класса в `_ClassName__value`

 **Пример**

Python

```
class Book:
    def __init__(self, title):
        self.__title = title  # приватный атрибут


book = Book("Brave New World")
# print(book.__title)     # AttributeError
print(book._Book__title)  # доступ через mangled-имя
```

Важно:

- Обращаться к атрибутам через `_ClassName__attr` **технически возможно**, но **делать так не рекомендуется** — это нарушает принцип инкапсуляции.

Защищённые и приватные методы

Методы, как и поля, могут быть:

- **Публичными** — доступны отовсюду
- **Защищёнными** — начинаются с одного подчёркивания: `_method()`
- **Приватными** — начинаются с двух подчёркиваний: `__method()`

Главная цель использования защищённых и приватных методов — **скрыть детали реализации**, которые **не предназначены для вызова снаружи**.

Защищённые методы

Защищённые методы используются, когда нужно **вынести часть логики в отдельный метод**, который **не должен вызываться напрямую извне**, но может быть **расширен в наследниках**.

```python
class Report:
    def __init__(self, data):
        self.data = data

    def generate(self):
        cleaned = self._prepare_data()
        print("Отчёт:")
        print("\n".join(cleaned))

    def _prepare_data(self):
        # Фильтрация пустых строк и обрезка пробелов
        return [line.strip() for line in self.data if line.strip()]

class SalesReport(Report):
    def _prepare_data(self):
        raw = super()._prepare_data()
        return [line for line in raw if not line.startswith("#")]  # удаляем
строки-комментарии

data = [
    "  Продажи за январь  ",
    "",
    " # внутренний комментарий ",
```

```
    "  Доход: $5000 "
]

report = SalesReport(data)
report.generate()
```

Особенности:

- `_prepare_data()` используется только **внутри класса**
- Его можно расширить в **дочерних классах**
- Но **внешнему коду** он не нужен — пользователь работает только с `generate()`

Приватные методы

Приватные методы применяются, когда часть логики должна быть **строго скрыта** и не должна вызываться извне или переопределяться в наследниках.

```python
class User:
    def __init__(self, name):
        self.name = name
        self.__id = self.__generate_id()

    def show_info(self):
        print(f"{self.name} - ID: {self.__id}")

    def __generate_id(self):
        # приватная логика генерации идентификатора
        from random import randint
        return f"user-{randint(1000, 9999)}"

user = User("Alice")
user.show_info()
# print(user.__generate_id())       # AttributeError
print(user._User__generate_id())    # доступ возможен, но нарушает инкапсуляцию
```

Особенности:

- `__generate_id()` — это **вспомогательная логика**, которая не нужна внешнему коду
- Метод **не может быть переопределён** в наследниках
- Пользователь взаимодействует только через `show_info()`, не зная деталей

IT CAREER HUB

Задания для закрепления 1

1. **Найдите ошибку в коде:**

```python
class Book:
    def __init__(self, title):
        self.__title = title

    def show_title(self):
        print(__title)

book = Book("1984")
book.show_title()
```

Посмотреть ответ

2. **Укажите верные утверждения об уровнях доступа в Python:**

a. Публичные атрибуты доступны везде
b. Защищённые атрибуты запрещено использовать вне класса
c. Приватные атрибуты создаются с одним подчёркиванием
d. Защищённые атрибуты могут использоваться в дочерних классах

Посмотреть ответ

IT CAREER HUB

Геттеры и сеттеры

Когда атрибут объявлен как **приватный** (например, `__price`), к нему нельзя обратиться **напрямую**.

Вместо этого создаются **специальные методы**, с помощью которых можно:

- **Прочитать значение** — геттер
- **Изменить значение** — сеттер

Такие методы позволяют **контролировать доступ** к полю и при необходимости — добавить **валидацию**, **логирование** или другие действия при чтении и записи.

Зачем использовать геттеры и сеттеры?

- **Безопасность** — можно защитить важные поля от некорректных изменений
- **Гибкость** — логика доступа может меняться, не изменяя внешний интерфейс
- **Поддержка инкапсуляции** — внутреннее устройство объекта остаётся скрытым

Пример

```python
class Temperature:
    def __init__(self):
        self.__celsius = 0  # приватное поле

    def get_celsius(self):
        return self.__celsius  # геттер

    def set_celsius(self, value):
        if value < -273.15:
            raise ValueError("Температура не может быть ниже абсолютного нуля")
        self.__celsius = value  # сеттер

temp = Temperature()
temp.set_celsius(25)       # установка значения
print(temp.get_celsius())  # чтение значения
```

```
# temp.set_celsius(-300)   # ValueError
```

Пример: установка значения при создании объекта

Иногда важно, чтобы поле было валидным уже в момент создания объекта. В этом случае проверку можно встроить в `__init__()`. Чтобы не дублировать проверку, удобно вынести логику в отдельный **приватный метод**.

Python

```python
class Temperature:
    def __init__(self, value):
        self.__validate_celsius(value)
        self.__celsius = value

    def get_celsius(self):
        return self.__celsius

    def set_celsius(self, value):
        self.__validate_celsius(value)
        self.__celsius = value

    @staticmethod
    def __validate_celsius(value):
        if value < -273.15:
            raise ValueError("Температура не может быть ниже абсолютного нуля")

temp1 = Temperature(20)
print(temp1.get_celsius())
# temp2 = Temperature(-500)  # ValueError
```

IT CAREER HUB

Декоратор @property

Когда необходимо **контролировать доступ к полям**, но при этом сохранить **читаемый и естественный синтаксис**, используется декоратор `@property`. Он позволяет обращаться к методам как к обычным атрибутам — например, `obj.value` вместо `obj.get_value()` — сохраняя при этом **гибкость и инкапсуляцию**.

Зачем нужен @property

- Чтобы **скрыть реализацию**, но предоставить понятный интерфейс (`temp.celsius`, а не `temp.get_celsius()`)
- Чтобы **контролировать доступ к полям** (например, проверять значение при установке)
- Чтобы сохранить возможность **менять внутреннюю структуру класса**, не меняя внешний синтаксис обращения

Синтаксис:

```python
class MyClass:
    def __init__(self, value):
        self.__attr = value

    @property  # всегда использовать property для геттера
    def attr(self):        # геттер
        return self.__attr

    @attr.setter  # использовать имя_геттера.setter для сеттера
    def attr(self, value):  # сеттер
        self.__attr = value
```

- `@property` используется для создания **геттера**
- `@attr.setter` используется для создания **геттера**
- Оба метода относятся к **одному "полю"** — `attr`

Пример: температура с валидацией

```python
class Temperature:
    def __init__(self, value):
        self.__celsius = 0
        self.celsius = value  # вызов сеттера внутри __init__

    @property
    def celsius(self):
        return self.__celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Температура не может быть ниже абсолютного нуля")
        self.__celsius = value

t = Temperature(25)
print(t.celsius)    # вызов геттера

t.celsius = 15      # вызов сеттера
print(t.celsius)    # вызов геттера

# t.celsius = -500  # ValueError: Температура не может быть ниже абсолютного
нуля
```

Read-only свойства

Иногда необходимо разрешить **только чтение значения**, но **запретить его изменение извне** — например, если значение вычисляется автоматически, или не должно меняться после создания объекта.

Для этого создаётся **только геттер с** `@property`, без сеттера. В этом случае свойство становится **только для чтения**.

IT CAREER HUB

Пример: температура, вычисляемая в Фаренгейтах

Python

```python
class Temperature:
    def __init__(self, celsius):
        self.__celsius = celsius  # приватное поле

    @property
    def celsius(self):
        # геттер для получения значения в цельсиях
        return self.__celsius

    @property
    def fahrenheit(self):
        # геттер вычисления значения в фаренгейтах
        return self.__celsius * 9 / 5 + 32

t = Temperature(25)
print(t.celsius)      # получение значения в фаренгейтах
print(t.fahrenheit)   # вычисление значения в фаренгейтах
# t.celsius = 100     # AttributeError
# t.fahrenheit = 100  # AttributeError
```

IT CAREER HUB

Задания для закрепления 2

1. **Найдите ошибку в коде:**

```python
class Temperature:
    def __init__(self, value):
        self.__celsius = value

    @property
    def __celsius(self):
        return self.__celsius

t = Temperature(25)
print(t.celsius)
```

Посмотреть ответ

IT CAREER HUB

Ответы на задания

Задания на закрепление 1	Вернуться к заданиям
1. Ошибка в коде	Ответ: в методе `show_title` нужно обращаться с помощью `self.__title`
2. Утверждения об уровнях доступа	Ответ: a, d
Задания на закрепление 2	Вернуться к заданиям
1. Ошибка в коде	Ответ: ошибка в названии метода — должно быть `celsius`, а не `__celsius`.

IT CAREER HUB

🔎 Практическая работа

1. Безопасная флешка

Создайте класс `SecureUSB`, представляющий защищённую флешку.

- При создании передаётся **секретное содержимое** и пароль
- Метод `unlock(password)` — возвращает `True`, если пароль верный, и разблокирует флешку
- Метод `lock()` — блокирует устройство
- Метод `read()` — возвращает сохранённые данные, если устройство разблокировано, иначе выбрасывает ошибку `PermissionError`.

Продумайте, какие поля следует скрыть, а какие оставить доступными.

Пример вывода:

```python
Device is locked.
Access denied.
Access granted.
Data: Secret plans
```

Решение:

```python
class SecureUSB:
    def __init__(self, secure_data, password):
        self.__secure_data = secure_data
        self.__password = password
        self.__locked = True

    def unlock(self, password):
        if password == self.__password:
            self.__locked = False
            return True
        return False

    def lock(self):
        self.__locked = True

    def read(self):
        if self.__locked:
            raise PermissionError("Device is locked. Access denied.")
        return self.__secure_data


usb = SecureUSB("Secret plans", "qwerty")

try:
    print(usb.read())
except PermissionError as e:
    print(e)

if not usb.unlock("1234"):
    print("Access denied.")

if usb.unlock("qwerty"):
    print("Access granted.")
data = usb.read()
print(f"Data: {data}")
```

2. Данные через свойство

Доработайте класс `SecureUSB`:

- Переделайте метод `read()` в свойство `data`, используя `@property`.
- Теперь доступ к содержимому осуществляется через обращение к `usb.data`, а не вызов метода.

При попытке чтения в заблокированном состоянии должно по-прежнему выбрасываться `PermissionError`.

Пример вывода:

```python
Device is locked. Access denied.
Access granted.
Data: Secret plans
```

IT CAREER HUB

Решение:

```python
class SecureUSB:
    def __init__(self, secure_data, password):
        self.__secure_data = secure_data
        self.__password = password
        self.__locked = True

    def unlock(self, password):
        if password == self.__password:
            self.__locked = False
            return True
        return False

    def lock(self):
        self.__locked = True

    @property
    def data(self):
        if self.__locked:
            raise PermissionError("Device is locked. Access denied.")
        return self.__secure_data


usb = SecureUSB("Secret plans", "qwerty")

try:
    print(usb.data)
except PermissionError as e:
    print(e)

if usb.unlock("qwerty"):
    print("Access granted.")
print(f"Data: {usb.data}")
```