

## Лекция 5

Здравствуйте, уважаемые слушатели!

Тема нашей лекции – Инкапсуляция и модификаторы доступа

План лекции:

1. Введение в инкапсуляцию
2. Основные принципы инкапсуляции
3. Модификаторы доступа
4. Преимущества инкапсуляции
5. Заключение

### 1. Введение в инкапсуляцию

Инкапсуляция — это один из основополагающих принципов объектно-ориентированного программирования (ООП), направленный на защиту данных и контроль доступа к ним. Основная идея инкапсуляции заключается в том, чтобы скрыть внутреннее устройство объекта, предоставляя доступ к данным и методам только через определенный интерфейс. Это позволяет предотвращать некорректное использование данных, а также упростить процесс изменения и поддержания кода.

Инкапсуляция создает своего рода «барьер» между внутренним состоянием объекта и внешним кодом, делая данные доступными только тем методам и функциям, которым они необходимы для выполнения задач. Этот принцип используется для повышения устойчивости кода, улучшения безопасности и обеспечения структурированности программ.

### 2. Основные принципы инкапсуляции

Инкапсуляция реализуется через скрытие данных и предоставление доступа к ним через методы. Основные принципы инкапсуляции включают:

1. **Скрытие данных:** Внутренние данные объекта (переменные, свойства) скрываются от доступа внешнего кода.
2. **Интерфейсный доступ:** Доступ к данным предоставляется через методы, которые контролируют возможность чтения и изменения этих данных.
3. **Защита целостности данных:** Методы могут включать проверки, чтобы предотвратить некорректное использование и изменение данных.

Реализация инкапсуляции зависит от модификаторов доступа, которые определяют, какие элементы класса будут доступны вне этого класса.

### 3. Модификаторы доступа

Модификаторы доступа — это механизмы, которые позволяют контролировать доступ к атрибутам и методам класса. С их помощью можно определять, кто и как будет иметь доступ к данным, тем самым обеспечивая защиту от непреднамеренного использования или изменения состояния объекта.

### 3.1 Типы модификаторов доступа

Существует несколько основных модификаторов доступа, которые используются для управления уровнем доступа к данным и методам:

- **Public (публичный):** Доступен из любого места программы. Это открытый интерфейс класса, который можно использовать извне.
- **Protected (защищенный):** Доступен только внутри класса и его подклассов. Это позволяет наследовать методы и свойства, но при этом ограничивает доступ к ним за пределами иерархии класса.
- **Private (приватный):** Доступен только внутри самого класса. Это наиболее строгий уровень доступа, который используется для инкапсуляции данных.

### 3.2 Модификаторы доступа в Python

В Python нет жесткой системы модификаторов доступа, как в некоторых других языках программирования. Однако, разработчики могут использовать соглашения об именовании для обозначения уровня доступа:

- **Публичные атрибуты и методы:** Обозначаются без начальных символов. Пример: `name`.
- **Защищенные атрибуты и методы:** Обозначаются одним подчеркиванием в начале. Пример: `_name`. Это соглашение говорит о том, что данный элемент не предназначен для использования вне класса и его подклассов.
- **Приватные атрибуты и методы:** Обозначаются двумя подчеркиваниями в начале. Пример: `__name`. Такие атрибуты и методы доступны только внутри класса, и их имена изменяются на уровне интерпретатора Python (`name mangling`), что ограничивает их доступ снаружи класса.

Пример с использованием разных уровней доступа:

```
python
```

```
Копировать код
```

```
class BankAccount:
```

```
    def __init__(self, owner, balance):
```

```
        self.owner = owner          # публичный атрибут
```

```
        self._account_number = "123456" # защищенный атрибут
```

```
self.__balance = balance # приватный атрибут

def deposit(self, amount):
    if amount > 0:
        self.__balance += amount
    return self.__balance

def withdraw(self, amount):
    if 0 < amount <= self.__balance:
        self.__balance -= amount
    return self.__balance

def get_balance(self):
    return self.__balance
```

В данном примере `owner` — публичный атрибут, доступный из любого места программы, `_account_number` — защищенный, а `__balance` — приватный.

## 4. Преимущества инкапсуляции

Инкапсуляция обеспечивает несколько важных преимуществ:

### 4.1 Повышение безопасности данных

Инкапсуляция позволяет ограничить доступ к критически важным данным, предотвращая их непреднамеренное или преднамеренное изменение. Это особенно важно при работе с финансовыми, личными и конфиденциальными данными.

### 4.2 Управление доступом к данным

Инкапсуляция позволяет разработчику определять, какие данные и методы будут доступны для использования, а какие останутся закрытыми. Например, можно разрешить чтение определенных данных, но запретить их изменение.

### 4.3 Упрощение изменения кода

Инкапсуляция позволяет изменять внутреннее представление данных без необходимости модифицировать внешний код, который использует класс. Поскольку взаимодействие с данными происходит через методы, любые изменения внутри класса остаются «прозрачными» для внешнего кода.

## 5. Методы для работы с инкапсулированными данными

Методы, предоставляющие доступ к инкапсулированным данным, играют важную роль в обеспечении контроля над доступом к этим данным.

## 5.1 Геттеры и сеттеры

Геттеры и сеттеры — это методы, которые позволяют контролировать доступ к приватным и защищенным атрибутам. Они используются для чтения и изменения данных, обеспечивая при этом возможность выполнения проверок и ограничений.

Пример геттеров и сеттеров:

```
python
Копировать код
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name

    def set_name(self, name):
        if isinstance(name, str):
            self.__name = name

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age
```

В этом примере `get_name` и `set_name` предоставляют доступ к приватному атрибуту `__name`, а `get_age` и `set_age` — к атрибуту `__age`. Это позволяет безопасно читать и изменять данные, проверяя их корректность.

## 5.2 Свойства (properties) в Python

Python предоставляет удобный способ работы с геттерами и сеттерами с помощью декоратора `@property`. Этот декоратор позволяет определять методы, которые могут использоваться как свойства.

Пример использования декоратора `@property`:

```
python
Копировать код
class Person:
    def __init__(self, name, age):
```

```
self.__name = name
self.__age = age
```

```
@property
def name(self):
    return self.__name
```

```
@name.setter
def name(self, value):
    if isinstance(value, str):
        self.__name = value
```

```
@property
def age(self):
    return self.__age
```

```
@age.setter
def age(self, value):
    if value > 0:
        self.__age = value
```

Здесь name и age используются как свойства, что позволяет обращаться к ним, как к обычным атрибутам, но при этом сохраняется возможность выполнять проверки и управлять доступом.

## 6. Примеры использования инкапсуляции

### 6.1 Управление банковским счетом

Инкапсуляция часто используется для создания классов, представляющих реальные объекты, такие как банковские счета. Например, мы можем создать класс BankAccount, который будет инкапсулировать баланс и ограничивать прямой доступ к нему:

```
python
Копировать код
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        return self.__balance
```

```

def withdraw(self, amount):
    if 0 < amount <= self.__balance:
        self.__balance -= amount
    else:
        print("Insufficient funds")
    return self.__balance

def get_balance(self):
    return self.__balance

```

Здесь `__balance` инкапсулируется и доступен только через методы `deposit`, `withdraw` и `get_balance`.

## 6.2 Контроль доступа к данным в системе управления студентами

Еще один пример использования инкапсуляции — управление данными студентов в учебной системе. Применяя инкапсуляцию, можно защитить личные данные студентов и предоставить доступ к ним только через интерфейс класса.

```

python
Копировать код
class Student:
    def __init__(self, name, student_id, grade):
        self.__name = name
        self.__student_id = student_id
        self.__grade = grade

    def get_student_info(self):
        return f"Student: {self.__name}, ID: {self.__student_id}"

    def set_grade(self, grade):
        if 0 <= grade <= 100:
            self.__grade = grade

    def get_grade(self):
        return self.__grade

```

Здесь `__name`, `__student_id` и `__grade` защищены от прямого доступа, и их значения могут быть изменены только через методы.

## 7. Преимущества и ограничения инкапсуляции

### 7.1 Преимущества

- **Безопасность данных:** Инкапсуляция позволяет скрыть внутренние данные и защитить их от некорректного использования.
- **Упрощение тестирования и отладки:** Инкапсулированные данные и методы легче тестировать, так как их использование ограничено определенным контекстом.
- **Гибкость и расширяемость:** Инкапсуляция позволяет изменять внутреннюю структуру класса без влияния на код, использующий этот класс.

## 7.2 Ограничения

- **Дополнительная сложность:** Инкапсуляция может увеличить сложность программы из-за необходимости создания геттеров и сеттеров.
- **Усложнение для начинающих:** Понимание принципов инкапсуляции и модификаторов доступа может быть затруднительным для начинающих разработчиков.

## 8. Заключение

Инкапсуляция и модификаторы доступа играют важную роль в обеспечении безопасности и устойчивости кода, предоставляя механизм для управления доступом к данным и сохранения целостности объектов. Защищая внутренние данные и методы от несанкционированного доступа и изменений, эти принципы позволяют разработчикам создавать надёжные системы, в которых объекты взаимодействуют друг с другом через строго определённые интерфейсы. Контроль доступа помогает избежать ошибок, связанных с некорректным использованием данных, и делает код более структурированным и лёгким для поддержки. Эти концепции являются важной частью объектно-ориентированного программирования, способствуя созданию стабильного и защищённого программного обеспечения.

Литературы:

1. Лекции по теории вероятностей и элементам математической статистики. Лекции по теории вероятностей. Пахнатов И. Москва-ЭМОСКО-2016 стр 34-41
2. Курс Упражнений по Теории Вероятностей и Математической Статистике. Серия Основ Математики Экономики. Версия-2 -2021 стр. 33-42