**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Доклад на тему:**

Принципы ООП

Выполнил:

Никифоров Арсен Германович

Группа: К3343d

Проверил:

Говоров Антон Игоревич

Санкт-Петербург

2023 г.

**Объектно-ориентированное программирование (ООП)** — это методология программирования, базирующаяся на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

ООП основывается на четырех основных принципах:

1. **Инкапсуляция**: Сокрытие деталей реализации и представление пользователям только функционального интерфейса.
2. **Наследование**: Создание новых классов на основе уже существующих.
3. **Полиморфизм**: Различное использование одного и того же метода в разных классах.
4. **Абстракция**: Выделение значимых характеристик объекта и игнорирование незначимых.

**Инкапсуляция** это подход в программировании, когда мы скрываем детали работы части кода от других частей программы. Мы показываем только то, что пользователь (или другая часть программы) должен видеть - функции, переменные и т.д. Таким образом, мы предоставляем функциональный интерфейс, не раскрывая детали реализации.

#include <iostream>

class Person **{**

private**:** // Пример инкапсуляции: скрытие деталей реализации

std**::**string name**;** // Приватное поле, доступное только внутри класса

public**:**

// Метод для установки имени

void setName**(**const std**::**string**&** newName**)** **{**

name **=** newName**;** // Изменение значения скрытого поля

**}**

// Метод для получения имени

std**::**string getName**()** const **{**

**return** name**;** // Возврат значения скрытого поля

**}**

**};**

int main**()** **{**

Person person**;** // Создание объекта класса Person

// Использование методов класса для работы с данными (инкапсуляция)

person**.**setName**(**"Арсен"**);** // Устанавливаем имя через метод

std**::**cout **<<** "Имя: " **<<** person**.**getName**()** **<<** std**::**endl**;** // Получаем имя через метод

**return** 0**;**

**}**

В данном примере:

* **Инкапсуляция** проявляется в том, что члены класса (name) объявлены как private, что означает, что они доступны только внутри самого класса и не доступны напрямую извне. Мы используем методы setName и getName для управления и получения значения переменной name, тем самым скрывая детали реализации (как хранится имя) от внешнего кода.

**Наследование** позволяет создавать новые классы на основе уже существующих с перениманием их свойств и методов. Это способствует повторному использованию кода.

\_\_\_\_

#include <iostream>

// Базовый класс (родительский класс)

class Animal **{**

public**:**

void sound**()** const **{** // Метод базового класса

std**::**cout **<<** "Животное издает звук" **<<** std**::**endl**;**

**}**

**};**

// Производный класс (наследник)

class Cat **:** public Animal **{** // Пример наследования: Cat наследует от Animal

public**:**

void purr**()** const **{** // Метод производного класса

std**::**cout **<<** "Кошка мурлычет" **<<** std**::**endl**;**

**}**

**};**

int main**()** **{**

Cat cat**;** // Создание объекта класса Cat

// Использование методов класса Cat (унаследованных и собственных)

cat**.**sound**();** // Мы можем использовать метод из базового класса (Animal)

cat**.**purr**();** // И метод из производного класса (Cat)

**return** 0**;**

**}**

В данном примере:

* **Наследование** проявляется в том, что класс Cat наследует от класса Animal. Это видно в строке class Cat : public Animal, где Cat - производный класс, а Animal - базовый класс. Производный класс (Cat) получает все члены (методы и переменные) базового класса (Animal), а также может добавить свои собственные члены (в данном случае, метод purr). Таким образом, Cat имеет доступ как к методу sound из Animal, так и к методу purr из Cat.

**Полиморфизм** в ООП используется для обеспечения возможности обработки объектов разных классов через общий интерфейс, что позволяет упростить и унифицировать код и расширять его функциональность.

#include <iostream>

// Базовый класс (родительский класс)

class Animal **{**

public**:**

virtual void sound**()** const **{**

std**::**cout **<<** "Животное издает звук" **<<** std**::**endl**;**

**}**

**};**

// Производный класс (наследник)

class Cat **:** public Animal **{**

public**:**

void sound**()** const override **{**

std**::**cout **<<** "Кошка мурлычет" **<<** std**::**endl**;**

**}**

**};**

// Функция, принимающая объекты базового класса

void makeSound**(**const Animal**&** animal**)** **{**

animal**.**sound**();** // Проявление полиморфизма: вызываем метод sound, который может иметь разное поведение в зависимости от типа объекта

**}**

int main**()** **{**

Animal genericAnimal**;** // Создание объекта базового класса

Cat cat**;** // Создание объекта производного класса

makeSound**(**genericAnimal**);** // Вызываем метод из базового класса

makeSound**(**cat**);** // Вызываем метод из производного класса (полиморфизм)

**return** 0**;**

**}**

В данном примере:

* **Полиморфизм** проявляется в методе sound(). В базовом классе Animal этот метод объявлен с ключевым словом virtual, что позволяет производным классам переопределить его. В классе Cat мы переопределили метод sound(), предоставив специфическое поведение для кошки. Функция makeSound принимает объекты базового класса Animal, и при вызове animal.sound() происходит вызов соответствующей реализации sound() для конкретного типа объекта, что является проявлением полиморфизма.

**Абстракция** в ООП используется для выделения основных характеристик и поведения объекта, скрывая детали его реализации, что позволяет упростить работу с объектами и сфокусироваться на ключевых аспектах.

#include <iostream>

class Cat **{**

private**:**

std**::**string name**;** // Абстракция: скрытие сложных деталей реализации

int age**;** // Абстракция: выделение ключевых характеристик объекта

public**:**

// Конструктор класса Cat, принимает имя и возраст

Cat**(**const std**::**string**&** catName**,** int catAge**)** **:** name**(**catName**),** age**(**catAge**)** **{}**

// Метод для представления кошки

void introduce**()** const **{**

std**::**cout **<<** "Меня зовут " **<<** name **<<** " и мне " **<<** age **<<** " лет." **<<** std**::**endl**;**

**}**

**};**

int main**()** **{**

Cat cat**(**"Барсик"**,** 3**);** // Создание объекта класса Cat

cat**.**introduce**();** // Вызываем метод для представления кошки

**return** 0**;**

**}**

В данном примере:

* **Абстракция** проявляется в том, что мы абстрагировались от множества деталей реализации кошки и оставили только важные характеристики: имя (name) и возраст (age). Мы не вдавались в технические детали того, как они хранятся или как работает метод introduce(). Вместо этого, мы сосредоточились на том, как использовать эту абстракцию, вызывая метод introduce(), чтобы представить кошку.

Абстракция позволяет упростить модель объекта, сконцентрировавшись на его важных характеристиках и функциях, а игнорируя лишние детали реализации.

**Заключение**

Принципы объектно-ориентированного программирования (ООП) предоставляют нам мощный инструментарий для структурирования и организации программ. Инкапсуляция, наследование, полиморфизм и абстракция помогают создавать более понятный, гибкий и легко поддерживаемый код. Эти принципы являются фундаментом ООП и используются в разработке множества программ и систем.