Въведение в ООП – какво е ООП, принципи на ООП, методи, член-данни, обекти, класове

Строга дефинция:

"**Object-oriented programming** (**OOP**) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as *attributes;* and code, in the form of procedures, often known as *methods.* A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another.[1][2] There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type."

Просто обяснение на обектно-ориентираното програмиране:

Нека разгледаме начинът по който използваме структури, с цел да обединяваме няколко различни типа данни в един обект, и функциите като начин да преизползваме един и същ код, с различни входни данни.

Ще разгледаме примерът с рационални числа. Ще напишем структура `Rational`, в която ще пазим рационални числа - числител и знаменател. Освен това, ще имаме функция, която ще събира две рационални числа.

```
#include <iostream>

struct Rational
{
    int numerator;
    int denominator;
};

Rational add(Rational a, Rational b)
{
    Rational c;
    if(a.denominator == b.denominator)
    {
        c.numerator = a.numerator + b.numerator;
        c.denominator = a.denominator;
    }
    else
    {
        c.numerator = a.numerator*b.denominator + b.numerator*a.denominator;
        c.denominator = a.denominator*b.denominator;
```

```
    }

    return c;

}
int main()
{
    Rational a,b,c;

    a.numerator = 5;
    a.denominator = 7;

    b.numerator = 3;
    b.denominator = 4;

    c = add(a,b);

    return 0;
}
```

В нашият случай, ние създадохме нов тип данна, `Rational`, и създадохме две "променливи" от тип `Rational`. Тези променливи, ще наричаме "обекти". От там идва и идеята за обектно-ориентирането програмиране. Вместо да имаме отделни променливи за числител и знаменател, ние създаваме обект от тип `Rational`. И всеки път, когато се нуждаем от това да използваме рационални числа в нашия код, ние можем да използваме `Rational`, без да се налага да го дефинираме всеки път.

В типа `Rational` ние имаме две променливи - `numerator` и `denominator` (числител и знаменател). Тези променливи ще наричаме **член-данни на `Rational`**

В дефиницията на ООП се спомена "методи". Методите са функции, които работят с данните. В нашият случай, функцията `add` е метод, с помощта на когото, ние работим (събираме) с член-данните на `Rational`.

tl;dr - обектите имат член-данни и методи, които работят с член-данните.

Класове

В дефиницията за ООП се спомена за **класове**

Какво е клас ?

Ние реализирахме типът `Rational`, с помощта на `struct`. Нека сега, вместо `struct`, използваме `Class`. Освен това, нека преместим методът `add()` вътре в класът `Rational`

```
#include <iostream>

class Rational
{
    private:
    int numerator;
    int denominator;
```

```
    public:
    void setNumerator(int n)
    {
        numerator = n;
    }
    int getNumerator()
    {
        return numerator;
    }
    void setDenominator(int n)
    {
        numerator = n;
    }
    int getDenominator()
    {
        return denominator;
    }

    Rational Rational::add(Rational b)
    {
        Rational c;
        if(this->denominator == b.denominator)
        {
            c.numerator = this->numerator + b.denominator;
            c.denominator =  this->denominator;
        }
        else
        {
            c.numerator = this->numerator*b.denominator + b.numerator*this->denominator;
            c.denominator = this->denominator*b.denominator;
        }

        return c;
    }
};


int main()
{
    Rational a,b,c;

    a.setNumerator(5);
    a.setDenominator(7);

    b.setNumerator(3);
    b.setDenominator(4);

    c = a.add(b);

    return 0;
}
```

Тук е важно да отбележим:

По подразбиране, в `struct`, член-данните (а и методите) са `public` (което е в разрез с принципът за енкапсулация, които ще разгледаме след малко), а при `class`, те са `private`, което довежда до компилационна грешка.

Защо използваме `class` вместо `struct`, ще кажем малко по-долу

Кои са основните принципи на ООП ?

ООП е изградено на базата на четири основни принципа - енкапсулация, абстракция, наследяване и полиморфизъм.

Енкапсулация

Да се върнем на примера с рационалните числа. Когато използваме `struct`, както в примера, ние имаме достъп до член-данните на `Rational`. Напълно е възможно да променим знаменателя на една дроб, което би довело до получаването на напълно друго число (3/4 не е равно на 3/2). Дори и да използваме `class`, където по подразбиране член-данните са `private`, пак не трябва да имаме директен достъп до тях, извън класът. Всякакви промени на член-данните трябва да бъдат регулирани по някакъв начин. Затова върху член-данните се прилага ключовата дума `private`, с цел да ограничим достъпа на тези член-данни само вътре в класът. Достъпа до тях, ще става чрез специални методи. Методите за промяна на стойността на някоя член-данна се нарича setter (мутатор), а методите за взимане на стойността на член-данните се нарича getter (селектор).

Тогава примерът с `Rational` се променя, като добавяме `getNumerator()`, `setNumerator()`, `getDenominator()`, `setDenominator()`, които имат модификатора `public`, с цел да може методите да са видим извън класът.

```
#include <iostream>

class Rational
{
    private:
    int numerator;
    int denominator;

    public:
    void setNumerator(int n)
    {
        numerator = n;
    }
    int getNumerator()
    {
        return numerator;
    }
    void setDenominator(int n)
    {
        numerator = n;
    }
    int getDenominator()
    {
        return denominator;
```

```
    }
};

Rational add(const Rational a, const Rational b)
{
    Rational c;
    if(a.getDenominator() == b.getDenominator())
    {
        c.setNumerator(a.getNumerator() + b.getNumerator());
        c.setDenominator(a.getDenominator());
    }
    else
    {
        c.setNumerator(a.getNumerator()*b.getDenominator() +
b.getNumerator()*a.getDenominator());
        c.setDenominator(a.getDenominator()*b.getDenominator());
    }

    return c;
}
int main()
{
    Rational a,b,c;

    a.setNumerator(5);
    a.setDenominator(7);

    b.setNumerator(3);
    b.setDenominator(4);

    c = add(a,b);

    return 0;
}
```

В този пример, `add()` е външна за класа `Rational`, и за това тя няма достъп до член-данните на `Rational`.

Абстракция

Да си представим, че имаме метод, който превръща нашето рационално число в такова, със запетая.

```
double convertToDouble(Rational a)
{
    return (double)(a.getNumerator() / a.getDenominator());
}
```

В този случаи, самото превръщане не е сложно. Важното е, че ако ползваме някъде `Rational` и съответно `convertToDouble()`, нас не ни интересува как е имплементиран този метод, за нас е важно, че него го има и че работи.

Нека разгледаме и друг пример - смартфонът. Когато натиснем някой бутон, съответното приложение се стартира. Ние не знаем как точно се случва това от гледна точка на кода, който седи зад това действие. За нас като разработчици е важно, да знаем, че ако потребителят натисне иконата на нашето приложение, то то ще се стартира. Ако разработчикът промени начина, по който се случва връзката между натискане на икона и стартиране на приложението, на нас това няма да ни се отрази - изходните точки - натискане на икона и стартиране на приложение са същите, са пътят между тях е различен.

tl;dr - Абстракцията ни позволява да "скриваме" детайли за имплементацията на дадени функции, като оставяме само входните и изходните точки видими

Наследяване

Много често се случва да имаме клас, които имат общи член-данни или методи с други класове. Съответно е бил зададен въпросът, може ли по някакъв начин да имаме клас, който да има общите черти, и да имаме класове, които да "наследяват" общите части, но да могат да добавят индивидуални неща.

В ООП, този проблем е разрешен, именно чрез наследяване. Класовете(child) могат да наследяват други обекти (parent), като "децата" "наследяват" всички член-данни и методи на родителите.

Пример

```
class Person
{
    char name[20]; //Име
    int age; //Възраст
}
```

```
class Student:Person
{
    int facultyNumber;
    double grades[10];
}
```

Нека `Student` наследява `Person`. Тогава `Student` освен `facultyNumber` и `grades`, той също има и член-данните `name` и `age`.

Полиморфизъм

Нека си представим, че имаме класове `Person`, `Student`, `Teacher`, `Staff`. За да бъде от полза нашата програма, която се грижи за огранизацията на хората в едно училище, то трябва да можеш да пазим информацията в един контейнер (например, масив). Но `Student` не може да е в един масив с `Teacher`, защото са различни класове. Полиморфизмът ни дава решението на този проблем. Всеки от `Student`, `Teacher`, `Staff`, може да се представи като `Person` и съответно можем да направим масив от `Person`. Освен че вече имаме масив, в който ще пазим всички хора, `Student` няма да изгуби специфичните за него член-данни или методи.

Наследяване и полиморфизъм ще бъдат разгледани в повече детайли по-късно в курса.
