Заштићени приступ

Службеним речима public и private за одређивање права приступа праве се две потпуно супротне могућности приступа: јавна поља и методе су доступне свима, док су приватна поља и методе неке класе доступни само тој класи.

Ове искључивости су довољне када се класе посматрају издвојено, али издојене класе не решавају сложене проблеме.

Обично је за основну класу корисно да изведеним класама омогући приступ неким од својих чланова, док истовремено сакрива те чланове од класа које нису део хијерархије наслеђивања.

У овом случају чланове можете да обележите службеном речи protected:

- Ако је класа A изведена из друге класе B, она може да приступа заштићеним члановима класе B
- Ако класа A није изведена из друге класе B, она не може да приступа ниједном заштићеном члану класе B; тако унутар класе A, заштићени члан класе B је у суштини приватан

У програмском језику C# методе и поља се могу декларисати као заштићени.

Али, већина смерница за ООП препоручује да поља у својим методама оставите тако да увек буду приватна, као и да одступите од овог ограничења само када је то неопходно.

Јавна поља угрожавају енкапсулацију пошто сви корисници клас еимају непосредан, неограничен приступ пољима.

Заштићена поља одржавају енкапсулацију за кориснике класе за које су заштићена поља недоступна.

Међутим, заштићена поља омогућавају да друге класе које наслеђују основну класу угрозе енкапсулацију.

Заштићеним члановима основне класе може да се приступи не само у изведеној класи него и у класама које су изведене из изведене класе.

Пример: дефинисати хијерархију класа за моделовање разних возила. Дефинисати основну класу Vozila и изведене класе Avion и Auto. Дефинисати заједничке методе под називима PokretanjeMotora и ZaustavljanjeMotora у класи Vozila и додати неке методе у обе изведене класе које су посебне за те класе. Додати виртуелну методу Pogon у класи Vozila и надјачати подразумевану примену те методе у обе изведене класе.

Користе се четири засебна фајла због четири класе: Program, Avion, Auto, Vozila

Program.cs

```
using System;
namespace Vozila
{
    class Program
    {
        static void realizacijaKoda()
        {
            Console.WriteLine("Putovanje avionom");
            Avion mojAvion = new Avion();
            mojAvion.PokretanjeMotora("Start motora");
```

```
            mojAvion.Uzletanje();
            mojAvion.Pogon();
            mojAvion.ZaustavljanjeMotora("Vrrr");

            Console.WriteLine("\nPutovanje autom");
            Auto mojAuto = new Auto();
            mojAuto.PokretanjeMotora("Brm brm");
            mojAuto.Ubrzavanje();
            mojAuto.Pogon();
            mojAuto.Kocenje();
            mojAuto.ZaustavljanjeMotora("Puc puc");

            Console.WriteLine("\nTestiranje polimorfizma");
            Vozila vozilo1 = mojAuto;
            vozilo1.Pogon();
            vozilo1 = mojAvion;
            vozilo1.Pogon();
        }

        static void Main()
        {
            try
            {
                realizacijaKoda();
            }
            catch (Exception izuzetak)
            {
                Console.WriteLine($"Izuzetak: {izuzetak.Message}");
            }
        }
    }
}
```

Vozila.cs

```
using System;
namespace Vozila
{
    class Vozila
    {
        public void PokretanjeMotora(string
                    zvukKojiSeCujePriStartovanju)
        {
            Console.WriteLine($"Pokretanje motora:
                    {zvukKojiSeCujePriStartovanju}");
        }
        public void ZaustavljanjeMotora(string
```

```
                    zvukKojiSeCujePriZaustavljanju)
        {
            Console.WriteLine($"Zaustavljanje Motora:
                    {zvukKojiSeCujePriZaustavljanju}");
        }
        public virtual void Pogon()
        {
            Console.WriteLine("Defoltna implementacija metoda Pogon");
        }
    }
}
```

Avion.cs

```
using System;
namespace Vozila
{
    class Avion : Vozila
    {
        public void Uzletanje()
        {
            Console.WriteLine("Uzletanje");
        }
        public void Sletanje()
        {
            Console.WriteLine("Sletanje");
        }
        public override void Pogon()
        {
            Console.WriteLine("Letenje");
        }
    }
}
```

Auto.cs

```
using System;
namespace Vozila
{
    class Auto:Vozila
    {
        public void Ubrzavanje()
        {
            Console.WriteLine("Ubrzavanje");
        }
        public void Kocenje()
        {
            Console.WriteLine("Kocenje");
```

```
        }
        public override void Pogon()
        {
            Console.WriteLine("Rad motora");
        }
    }
}
```

Даје:
Putovanje avionom
Pokretanje motora: Start motora
Uzletanje
Letenje
Zaustavljanje Motora: Vrrr

Putovanje autom
Pokretanje motora: Brm brm
Ubrzavanje
Rad motora
Kocenje
Zaustavljanje Motora: Puc puc

Testiranje polimorfizma
Rad motora
Letenje

Све класе које се изводе из класе Vozila наслеђују њене методе.
Вредности за параметре ће бити различите за све типове возила и помоћи ће да се касније препозна које возило се покреће и зауставља.
Оба начина превоза позивају подразумевану примену виртуелне методе Pogon.
Код realizacijaKoda проверава полиморфизам који нуди метода Pogon.
Код прави референцу на објекат типа Auto коришћењем промењиве Vozila а затим позива методу Pogon коришћењем промењиве Vozila.
Метода Pogon је виртуелна, тако да извршни модул одређује коју верзију методе Pogon треба да позове преко промењиве Vozila на основу стварног типа објекта на који указује та промењива.

<u>Креирање проширених метода</u>
Наслеђивање је алат који омогућава да се проширује функционалност класе креирањем нове класе која се изводи из те класе.
Међутим, понекад коришћење наслеђивања није најпогоднији механизам за додавање нових понашања, посебно ако вам је потребно да брзо проширите неки тип без утицаја на постојећи код.
Нпр, ако желимо да додамо нове могућности на тип int, као што је метода под називом Negacija, која враћа негативну вредност коју неки цео број тренутно садржи.

Један од начина да се то постигне је да се дефинише нови тип под називом NegInt32, који наслеђује тип System.Int32 (тип int је синоним за тип System.Int32) и да се дода метода Negacija:

```
class NegINt32: System.Int32
{
     public int Negacija()
     {
          //...
     }
}
```

Теоретски, тип NegInt32 ће наследити све функционалности повезане са типом System.Int32, уз додатак методе Negacija.

Постоје два разлога зашто овај приступ није добар:

- Ова метода се примењује само за тип NegInt32 а ако бисте желели да је користите са постојећим промењивима типа int у коду, морлаи бисте да промените дефиницију свих промењивих типа int у тип NegInt32
- Тип System.Int32 је заправо структура а не класа, а наслеђивање се не користи у структурама

Зато проширене методе постају корисне.

Коришћењем проширене методе може се проширити постојећи тип (класа или структура) додатним статичким методама.

Ове статичке методе су одмах доступне свим исказима у коду који указују на податке типа који се проширује.

Проширена метода се дефинише у статичкој класи и наведе се тип за који се одређена метода примењује као први параметар методе, заједно са службеном речи this.

Пример показује како би се могло применити проширена метода Negacija за тип int:

```
public static int Negacija(this int i)
{
     return -i;
}
```

Овде се this користи као префикс помоћу којег се Negacija препознаје као проширена метода и чињеница да је параметар са префиксом this типа int значе да проширују тип int.

```
using System;
namespace ProjekatCS002
{
    static class Util
    {
        public static int Negacija(this int i)
        {
            return -i;
        }
    }
    public class Program
    {
        public static void Main()
        {
```

```
            int x = 591;
            Console.WriteLine($"x.Negacija {x.Negacija()}");
        }
    }
}
```

Види се да није потребно у исказу који позива методу Negacija указати на класу Util.
Компајлер аутоматски препознаје све проширене методе за дати тип из свих статичких класа које су у области важења.
Методу Util.Negacija можемо да позивамо и прослеђивањем вредности типа int као параметар:

```
Console.WriteLine($"x.Negacija {Util.Negacija(x)}");
```

Пример: додавање проширене методе типу int; са том проширеном методом претварати вредност промењиве типа int са основом 10 на приказ те вредности са другачијом бројном основом

Program.cs

```
using System;
using Extensions;
namespace ExtensionMethod
{
    class Program
    {
        static void RealizacijaKoda()
        {
            int x = 10;
            for(int i=2; i<=10; i++)
            {
                Console.WriteLine($"{x} sa osnovom {i} je
                          {x.ConvertToBase(i)}");
            }
        }
        static void Main()
        {
            try
            {
                RealizacijaKoda();
            }
            catch (Exception izuzetak)
            {
                Console.WriteLine($"Izuzetak: {izuzetak.Message}");
            }
        }
    }
}
```

Util.cs

```
using System;
namespace Extensions
{
    static class Util
    {
        public static int ConvertToBase(this int i, int
               bazaUKojuSeKonvertuje)
        {
            if (bazaUKojuSeKonvertuje<2||bazaUKojuSeKonvertuje>10)
            {
                throw new ArgumentException("Vrednost se ne moze
                     konvertovati u osnovu " +
                bazaUKojuSeKonvertuje.ToString());
            }

            int rezultat = 0;
            int iteracije = 0;
            do
            {
                int sledecaCifra = i % bazaUKojuSeKonvertuje;
                i /= bazaUKojuSeKonvertuje;
                rezultat += sledecaCifra * (int)Math.Pow(10,
                     iteracije);
                iteracije++;
            }
            while (i != 0);
            return rezultat;
        }
    }
}
```

Даје:

10 sa osnovom 2 je 1010
10 sa osnovom 3 je 101
10 sa osnovom 4 je 22
10 sa osnovom 5 je 20
10 sa osnovom 6 je 14
10 sa osnovom 7 je 13
10 sa osnovom 8 je 12
10 sa osnovom 9 je 11
10 sa osnovom 10 je 10