Programowanie obiektowe – C#

DZIEDZICZENIE, POLIMORFIZM, BASE, METODY WIRTUALNE, ABSTRACT, SEALED

Klasa bazowa i pochodna

Klasa może dziedziczyć z jednej klasy, ale może implementować wiele interfejsów. Składnia dziedziczenia:

```
modyfikator_dostepu class klasa_bazowa
{
    …
}
class klasa_pochodna : klasa_bazowa
{
    …
}
```

Przykład

```
class Program
    {
        static void Main(string[] args)
        {
            Prostokat pr = new Prostokat();
            pr.UstawSzerokosc(4);
            pr.UstawWysokosc(5);

            // Obliczenie powierzchni
            Console.WriteLine("Powierzchnia prostokąta: {0}", pr.ObliczPowirzchnie());
            Console.ReadKey();

            // Wynik działania programu
            // Powierzchnia prostokata: 20
        }
    }

    // klasa bazowa
    class Ksztalt
    {
        // modyfikator dostepu protected
        // pola dostepne sa dla klasy oraz klas, której po niej dziedziczą
        // gdybyśmy zastosowali modyfikator dostępu private
        // pole byłoby dostępne tylko dla tej klasy
        protected int szerokosc;
        protected int wysokosc;

        public void UstawWysokosc(int w)
        {
            wysokosc = w;
        }
        public void UstawSzerokosc(int s)
        {
            szerokosc = s;
        }
    }

    // klasa pochodna
    class Prostokat : Ksztalt
    {
        public int ObliczPowirzchnie()
        {
            // mamy dostęp do pól z klasy bazowej
            return wysokosc * szerokosc;
```

```
        }
    }
```

Polimorfizm

Pojęcie polimorfizmu w języku C# jest związane z dziedziczeniem. Polimorfizm (gre. Polýmorphos wielopostaciowy) oznacza możliwość operowania na obiektach należących do różnych klas.

```
class Vehicle
{
    public void TurnLeft()
    {
        Console.WriteLine("Hello it's TurnLeft() from Vehicle");
    }
}
class Car : Vehicle
{
    public void TurnLeft()
    {
        Console.WriteLine("Hello it's TurnLeft() from Car");
    }
}
```

Klasa Car dziedziczy po Vehicle. Obie posiadają metodę TurnLeft(). Jeśli utworzymy nową instancję klasy Car i wywołamy metodę TurnLeft(), wykonany zostanie kod z klasy Car. Kompilator podpowie nam jednak że:

'Car.TurnLeft()' hides inherited member 'Vehicle.TurnLeft()'. Use the new keyword if hiding was intended

Chociaż z punktu widzenia kompilatora nie jest to błąd (taki kod zostanie skompilowany), dobrą praktyką jest jawne określenie, że wiemy, co robimy, i chcemy przykryć element uprzednio zadeklarowany w klasie bazowej. Służy do tego słowo kluczowe new. W tym kontekście to słowo służy jako modyfikator dostępu:

```
public new void TurnLeft()
    {
        Console.WriteLine("Hello it's TurnLeft() from Car");
    }
```

Za sprawą modyfikatora new komunikat informujący o przykrywaniu metody nie będzie się już więcej pokazywał. Informujemy tym samym kompilator, iż wiemy, że w klasie bazowej istnieje metoda TurnLeft(), ale chcemy zadeklarować metodę o takiej samej nazwie w klasie potomnej.

Takie działanie nie dotyczy jedynie metod klas, ale wszystkich elementów, włączając właściwości i pola.

Podobnie możemy przykrywać statyczne elementy

```
public class Vehicle
{
    public static int ACTUAL_MILEAGE = 100;
    public static void GetMileage()
    {
        Console.WriteLine($"Mileage {ACTUAL_MILEAGE}");
    }
}
public class Car : Vehicle
```

```
{
    new public static readonly int ACTUAL_MILEAGE = 1000;
    public new static void GetMileage()
    {
        Console.WriteLine($"Mileage {ACTUAL_MILEAGE}");
    }
}
```

Wykorzystanie:

```
Vehicle.ACTUAL_MILEAGE = 1;
Car.ACTUAL_MILEAGE = 1; //ERROR
```

BASE

Słowo kluczowe base ma bardzo ważne znaczenie w kontekście polimorfizmu ponieważ pozwala nam na uzyskanie dostępu do elementów klasy bazowej:

```
public class Vehicle
{
    public int ACTUAL_MILEAGE = 100;
    public void GetMileage()
    {
        Console.WriteLine($"Mileage {ACTUAL_MILEAGE}");
    }
}
public class Car : Vehicle
{
    new public readonly int ACTUAL_MILEAGE = 1000;
    public void GetInfo()
    {
        base.GetMileage();
    }
}
```

Bardziej zaawansowane wykorzystanie słówka kluczowego base mamy w przypadku konstruktorów

```
public class Vehicle
{
    private string _manufacturer;
    public Vehicle() => _manufacturer = "BMW";
    public Vehicle(string manufacturer) => _manufacturer = manufacturer;
    public string Manufacturer
    {
        get
        {
            return _manufacturer;
        }
        set
        {
            _manufacturer = value;
        }
    }
}
public class Car : Vehicle
{
    private string _model;
    public Car() => _model = "????";
    public Car(string model) : base("AUDI") => _model = model;
    public Car(string manufacturer, string model) : base(manufacturer) => _model = model;
    public string Model
    {
        set
        {
            _model = value;
        }
    }
```

```
    public string FullName
    {
        get
        {
            return $"{Manufacturer} {_model}";
        }
    }
}

var item1 = new Vehicle();
var item2 = new Vehicle("OPEL");
var item3 = new Car();
var item4 = new Car("A5");
var item5 = new Car("KIA", "POLICYJNA");
Console.WriteLine(item1.Manufacturer);
Console.WriteLine(item2.Manufacturer);
Console.WriteLine(item3.FullName);
Console.WriteLine(item4.FullName);
Console.WriteLine(item5.FullName);
```

OUTPUT:

BMW

OPEL

BMW ????

AUDI A5

KIA POLICYJNA

Metody virtualne

Metoda wirtualna to taka, która jest przygotowana do zastąpienia w klasie potomnej.

```
public class Vehicle
{
    public virtual void TurnLeft() //metody virutal nie mogą być private
    {
        Console.WriteLine("GO GO GO");
    }
}
```

Przedefiniowanie (ang. override) to proces polegający na tworzeniu nowej wersji metody w klasie potomnej. Polega on na utworzeniu metody, która opatrzona będzie słowem kluczowym override:

```
public class Car : Vehicle
{
    public override void TurnLeft()
    {
        Console.WriteLine("Im Going left");
        base.TurnLeft();
    }
}
public class Train : Vehicle
{
    public override void TurnLeft()
    {
        Console.WriteLine("How can I turn left??? ( ͡° ͜ʖ ͡°)");
    }
}
```

Program

```
static void Main(string[] args)
{
    Vehicle vehicle;
    Console.WriteLine("1-Car; 2-Train; def-Vehicle");
    var key = Console.ReadLine();
    switch (key)
    {
        case "1":
            vehicle = new Car();
            break;
        case "2":
            vehicle = new Train();
            break;
        default:
            vehicle = new Vehicle();
            break;
    }
    vehicle.TurnLeft();
    Console.ReadLine();
}
```

Output

Dla 1:

Im Going left

GO GO GO

Dla 2:

How can I turn left??? (͡° ͜ʖ ͡°)

Def:

GO GO GO

Gdy na samym początku deklarujemy „Vehicle vehicle;" określamy, iż wszelkie wywołania metody TurnLeft() będą odnosić się właśnie do tej klasy. Jest to tzw. wczesne powiązanie (ang. early binding). Jeżeli zastosujemy metody wirtualne, kompilator wstrzyma się z decyzją, do jakiej klasy przypisać daną metodę. Ta decyzja zostanie podjęta dopiero w trakcie działania programu. W powyższym naszym programie podejmuje ją użytkownik przy pomocy klawiszy 1, 2 lub inny. Takie rozwiązanie nazywamy późnym powiązaniem (ang. late binding).

Abstract

W poprzednich przykładach mieliśmy klasę bazową Vehicle, zawierającą metodę TurnLeft(). Ale jeśli założymy, że w programie nie będziemy korzystać z klasy Vehicle, a z klas potomnych, nie ma potrzeby implementowania metod w tej klasie. Możemy wtedy skorzystać ze słówka kluczowego abstract. Klasa abstrakcyjna nie ma implementacji (tj. definicji metod), ma jedynie nagłówki (deklaracje):

```
abstract class Vehicle
{
    public abstract void TurnLeft(); //Metody abstract nie moga byc private
}
```

Wtedy klasy potomne muszą mieć zaimplementowaną metodę abstrakcyjną:

```
public class Car : Vehicle
{
    public override void TurnLeft()
    {
```

```
        Console.WriteLine("Im Going left");
    }
}
public class Train : Vehicle //ERROR
{
}
```

Sealed

Sealed – elementy zabezpieczone (zaplombowane) nie mogą służyć jako klasy bazowe. Używamy tylko w przypadku, gdy mamy do tego poważny powód (np. względy bezpieczeństwa). Należy wtedy pamiętać, aby nie korzystać z metod wirtualnych ani z modyfikatora protected, bo nie ma to najmniejszego sensu.

```
public sealed class Vehicle
{
}
```

Zadania do samodzielnego rozwiązania:

1. Zaimplementuj klasę Shape, posiadającą właściwości X, Y, Height, Width oraz virutalną metodę Draw. Następnie zaimplementuj klasy:

- Rectangle,
- Triangle,
- Circle

Które będą implementować metodę draw poprzez wypisanie na okno konsoli jaką figurę próbujemy narysować.

Następnie napisz program, który do listy List<Shape>, doda po obiekcie każdego typu z klas dziedziczących. Następnie wywołaj dla każdego elementu w liście funkcję draw.

2. Napisz program wykorzystując polimorfizm, który będzie sprawdzał czy nauczyciel może wypuścić do domu uczniów swojej klasy bez opieki dorosłego:

Generator pesel: https://pesel.cstudios.pl/O-generatorze/Generator-On-Line

1) Utwórz projekt, w którym:

- Zdefiniujesz klasę virutalną Osoba o polach:
    - Imię
    - Nazwisko
    - Pesel
- • Zdefiniujesz metody:
    - SetFirstName
    - SetLastName
    - SetPesel
    - GetAge
    - GetGender //Pozycja 10 (kobiety parzyste, mezczyzni nieparzyste)
- Zdefiniujesz metody

- GetEducationInfo
- GetFullName
- CanGoAloneToHome

2) Dodaj klasę Uczen, która dziedziczy po klasie Osoba;

- Zawiera dodatkowe właściwości:
  - Szkoła
  - MozeSamWracacDoDomu
- Zdefiniujesz metody:
  - SetSchool
  - ChangeSchool
  - SetCanGoHomeAlone
- Implementuje zadeklarowane metody z klasy Osoba
  - Info - Nie może sam wracać poniżej 12 lat chyba że ma pozwolenie

3) Dodaj klasę Nauczyciel, która dziedziczy po klasie Uczen:

- Zawiera dodatkowe właściwości:
  - TytulNaukowy
  - Kolekcja uczniów -PodwladniUczniowie (uczniowie którzy znajdują się w klasie nauczyciela)
- Zadefiniujemy metody:
  - WhichStudentCanGoHomeAlone(Datetime dateToCheck) [wypisuje nazwiska uczniow ktorzy moga isc sami do domu]