Politechnika Wrocławska
Wydział Mechaniczny
Katedra Konstrukcji i Badań Maszyn i Pojazdów

Programowanie obiektowe Lab 9

Dr inż. Paweł Maślak

pawel.maslak@pwr.edu.pl

Laboratorium 9: Dziedziczenie Polimorfizm

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:

```
using System;

namespace Dziedziczenie
{
    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;
        }
    }
}
```

Odwołanie do klasy bazowej

Klasa pochodna dziedziczy składowe klasy bazowej, tj. pola, metody. Podczas dziedziczenia wielokrotnie pojawi się potrzeba uzyskania dostępu do składowych klasy bazowej. Dostęp do takich pól czy metod jest możliwy po użyciu słowa kluczowego base. Może ono zostać również użyte do przekazania parametrów konstruktora do klasy bazowej. Poniżej przykład, który pozwoli lepiej zrozumieć regułu użycia słowa kluczowego base:

```
using System;

namespace DziedziczenieBazowa
{
    class Program
    {
        static void Main(string[] args)
        {
            Blat tp = new Blat(4, 5);
            tp.WyswietInformacje();
            Console.ReadKey();

            // Wynik działania programu
            //Dlugosc: 4
            //Szerokosc: 5
            //Powierzchnia: 20
            //Koszt: 1000
        }
    }

    // klasa bazowa
    class Prostokat
    {
        protected int dlugosc;
        protected int szerokosc;

        public Prostokat(int d, int s)
        {
            dlugosc = d;
            szerokosc = s;
        }

        public int ObliczPowierzchnie()
        {
            return dlugosc * szerokosc;
        }

        public void WyswietlInformacje()
        {
            Console.WriteLine("Długość: {0}", dlugosc);
            Console.WriteLine("Szerokość: {0}", szerokosc);
            Console.WriteLine("Powierzchnia: {0}", ObliczPowierzchnie());
        }
    }

    // klasa pochodna
    class Blat : Prostokat
    {
  // Słowo kluczowe base przy konstruktorze pozwala nam wywowołać konsturktor klasy bazowej
  // W tym momencie przekazaliśmy nasze parametry do konstruktora klasy bazowej
```

```
        public Blat(int d, int s) : base(d,s)
        {
        }

        public int Koszt()
        {
            int koszt;
            koszt = ObliczPowierzchnie() * 50;
            return koszt;
        }

        public void WyswietInformacje()
        {
        // słowo kluczowe base pozwala nam odwołać się do składowych klasy bazowej
        // dla kompilatora ważniejsze się zmienne z klasy w której właśnie jesteśmy
        // za pomocą słowa kluczowe base wskazujemy jednoznacznie do której składowej
        // chcemy się odwołać. Dziękimi poniższemu wywołaniu w obecnej metodzie wywołamy
        // również metodę z klasy bazowej - wyświetlona zostanie większa ilość informacji
            base.WyswietlInformacje();
            Console.WriteLine("Koszt: {0}", Koszt());
        }

    }
}
```

Wielokrotne dziedziczenie

Język C# nie obsługuje wielokrotnego dziedziczenia. Klasa może dziedziczyć po jednej klasie bazowej, ale może implementować wiele interfejsów

```
using System;

namespace DziedziczenieInterfejsy
{
    class Program
    {
        static void Main(string[] args)
        {
            Prostokat pr = new Prostokat(4, 5);
            Console.WriteLine(pr.WyswietDlugosc(pr.dlugosc));
            Console.WriteLine(pr.WyswietSzerokosc(pr.szerokosc));
            Console.WriteLine("Cena to: {0}", pr.ObliczKoszt(25));

            Console.ReadKey();
            // Wynik działania programu
            //Dlugosc to: 4
            //Szerokosc to: 5
            //Cena to: 500
        }
    }

    // klasa bazowa
    class Ksztalt
    {
        public int dlugosc;
        public int szerokosc;

        public Ksztalt(int d, int s)
        {
            dlugosc = d;
            szerokosc = s;
        }

        public int ObliczPowierzchnie()
        {
            return dlugosc * szerokosc;
        }
    }

    // definicja interfejsu
    // zawiera tylko szkielet metody
    public interface ObliczKoszt
    {
        int ObliczKoszt(int powierzchnia);
    }

    public interface WyswietlanieInformacji
    {
        string WyswietDlugosc(int dlugosc);

        string WyswietSzerokosc(int szerokosc);
    }

    class Prostokat : Ksztalt, ObliczKoszt, WyswietlanieInformacji
    {
```

```
        public Prostokat(int d, int s) : base(d, s)
        {
        }
        // implementacja metody interfejsu ObliczKoszt
        public int ObliczKoszt(int p)
        {
            int koszt;
            koszt = p * ObliczPowierzchnie();
            return koszt;
        }

        // implementacja metod interfejsu WyswietlanieInformacji
        public string WyswietDlugosc(int dlugosc)
        {
            string info = String.Format("Długość to: {0}", dlugosc);
            return info;
        }

        public string WyswietSzerokosc(int szerokosc)
        {
            string info = String.Format("Szerokość to: {0}", szerokosc);
            return info;
        }
    }
}
```

Polimorfizm

Polimorfizm może być **statyczny** i **dynamiczny**.

- W polimorfizmie statycznym odpowiedź funkcji jest określna w trakcie kompilowania.
- W polimorfizmie dynamicznym odpowiedź ta jest podejmowana w czasie wykonywania programu.

Polimorfizm statyczny

Mechanim łączenia metody z obiektem w trakcie kompilacji jest nazywany wczesnym wiązaniem lub też statycznym wiązaniem. C# udostępnia dwa sposoby implementowania statycznego polimorfizmu:

- przeciążanie metod;
- przeciążanie operatorów.

Przeciążanie metod

W tej samej definicji klasy może znajdować się wiele funkcji o tej samej nazwie. Definicja metod musi się różnić od siebie typem i/lub liczbą parametrów. Nie można przeciążyć metod, które różnią się tylko zwracanym typem.

```
using System;
namespace PolimorfizmPrzeciazanieMetod
{
    class Program
    {
        static void Main(string[] args)
        {
            WyswietlanieDanych wd = new WyswietlanieDanych();
            wd.Wyswietl(4);
            wd.Wyswietl(4.5);
            wd.Wyswietl("4.5");
            Console.ReadKey();
            // Wynik działania programu
            //Wyswietlana liczba: 4
            //Wyswietlana liczba: 4.5
            //Wyswietlany tekst: 4.5
        }
    }
    class WyswietlanieDanych
    {
        public void Wyswietl(int i)
        {
            Console.WriteLine("Wyswietlana liczba: {0}", i);
        }
        public void Wyswietl(double d)
        {
            Console.WriteLine("Wyswietlana liczna: {0}", d);
        }
        public void Wyswietl(string s)
        {
```

```
            Console.WriteLine("Wyswietlany tekst: {0}", s);
        }
    }
}
```

Przeciążanie operatorów

C# pozwala na zmianę lub przeciążenie większości wbudowanych operatorów. Programista może używać operatorów z typami zdefiniowanymi również przez użytkownika. Przeciążone operatory to metody z nazwą, słowem kluczowym operator, po którym występuje symbol operatora, który chcemy zdefiniować. Przeciążony operator ma typ zwracany oraz listę parametrów.

Przejdziemy od razu do obszernego przykładu gdyż tak łatwiej będzie zrozumieć jak w praktyce wygląda przeciążanie operatorów.

```
using System;
namespace PolimorfizmPrzeciazanieOperatorow
{
    class Program
    {
        static void Main(string[] args)
        {
            double objetosc = 0;
            Pudelko p1 = new Pudelko();
            Pudelko p2 = new Pudelko();
            Pudelko p3 = new Pudelko();
            // specyfikacja 1
            p1.PobierzDlugosc(3.5);
            p1.PobierzSzerokosc(4.0);
            p1.PobierzWysokosc(5.5);
            // specyfikacja 2
            p2.PobierzDlugosc(2.5);
            p2.PobierzSzerokosc(5.0);
            p2.PobierzWysokosc(4.5);
            // specyfikacja 3
            p3.PobierzDlugosc(12.5);
            p3.PobierzSzerokosc(15.0);
            p3.PobierzWysokosc(14.5);
            // objetosc 1
            objetosc= p1.ObliczObjetosc();
            Console.WriteLine("Objetosc 1: {0}", objetosc);
            // objetosc 2
            objetosc = p2.ObliczObjetosc();
            Console.WriteLine("Objetosc 2: {0}", objetosc);
            // Dodanie 2 obiektów
            p3 = p1 + p2;
            // objetosc 3
            objetosc = p3.ObliczObjetosc();
            Console.WriteLine("Objetosc 3: {0}", objetosc);
            Console.ReadKey();
            // Wynik działania programu
            //Objetosc 1: 77
            //Objetosc 2: 56.25
            //Objetosc 3: 540
        }
    }
```

```
    class Pudelko
    {
        private double dlugosc;
        private double szerokosc;
        private double wysokosc;
        public void PobierzDlugosc(double d)
        {
            dlugosc = d;
        }
        public void PobierzSzerokosc(double s)
        {
            szerokosc = s;
        }
        public void PobierzWysokosc(double w)
        {
            wysokosc = w;
        }
        public double ObliczObjetosc()
        {
            return (dlugosc * szerokosc * wysokosc);
        }
        // Przeciążenie operatora +
        // Dodanie do siebie dwóch typów
        public static Pudelko operator+(Pudelko a, Pudelko b)
        {
            Pudelko pud = new Pudelko();
            pud.wysokosc = a.wysokosc + b.wysokosc;
            pud.szerokosc = a.szerokosc + b.szerokosc;
            pud.dlugosc = a.dlugosc + b.dlugosc;
            return pud;
        }
    }
}
```

Polimorfizm dynamiczny

C# pozwala tworzyć klasy abstrakcyjne, które następnie są implementowane w klasach pochodnych. Klasa taka zawiera abstrakcyjne metody, których implementacja zależy od wykorzystania w poszczególnych klasach pochodnych.

Poniżej lista zasad o których należy pamiętać tworząc klasy abstrakcyjne:

- nie można utworzyć instancji klasy abstrakcyjnej;
- nie można zadeklarować metody abstrakcyjnej poza klasą abstrakcyjną;
- kiedy klasa opatrzona jest modyfikatorem dostępu **sealed** nie może być dziedziczona. Dodatkowo, klasa abstrakcyjna nie może być zdefinowana jakas **sealed**.

```
using System;
namespace PolimorfizmKlasaAbstrakcyjna
{
    class Program
    {
        static void Main(string[] args)
        {
            Kwadrat kw = new Kwadrat(4,5);
            double pow = kw.Powierzchnia();
            Console.WriteLine("Powierzchnia figury: {0}", pow);
            Console.ReadKey();
            // Wynik działania programu
            //Powierzchnia figury: 20
        }
    }
    abstract class Ksztalt
    {
        public abstract int Powierzchnia();
    }
    class Kwadrat : Ksztalt
    {
        // klasa pochodna musi implementować metody klasy bazowej
        private int wysokosc;
        private int szerokosc;
        public Kwadrat(int a, int b)
        {
            wysokosc = a;
            szerokosc = b;
        }
        public override int Powierzchnia()
        {
            return (wysokosc * szerokosc);
        }
    }
}
```

Jeżeli masz zdefiniowaną metodę w klasie bazowej, ale chcesz, żeby została zaimplementowana w klasach pochodnych możesz do tego celu zastosować metody virtualne. Metody te mogą mieć różne implementacje w klasach pochodnych, ale nie muszą. Jeżeli w klasie pochodnej nie będzie implementacji metody wirtualnej z klasy bazowej to użyta zostanie

domyślna implementacja z klasy bazowej. Wybór odnośnie wywołania metody podejmowany jest w czasie wykonywania programu.

Polimorfizm dynamiczny jest realizowany za pomocą **klas abstrakcyjnych** oraz **metod wirtualnych.**

```
using System;
namespace PolimofrizmMetodyWirtualne
{
    class Program
    {
        static void Main(string[] args)
        {
            WywolanieKlass wk = new WywolanieKlass();
            Prostokat pr = new Prostokat(4, 5);
            Trojkat tr = new Trojkat(4, 5);
            BezImplementacji bi = new BezImplementacji(4, 5);
            wk.WywolajKlase(pr);
            wk.WywolajKlase(tr);
            wk.WywolajKlase(bi);
            // Zamiast powyższego wywołnia można byłoby użyć poniższego zapisu:
            //pr.Powierzchnia();
            //tr.Powierzchnia();
            //bi.Powierzchnia();
            // chciałem jednak pokazać możliwości, jakie niesie za sobą dziedziczenie
            Console.ReadKey();
            // Wynik działania programu
            //Powierzchnia kwadratu:
            //Powierzchnia: 20
            //Powierzchnia trójkata:
            //Powierzchnia: 10
            //Domyslna powierzchnia figury:
            //Powierzchnia: 0
        }
    }
    class Ksztalt
    {
        protected int wysokosc, szerokosc;
        public Ksztalt(int a, int b)
        {
            wysokosc = a;
            szerokosc = b;
        }
        public virtual int Powierzchnia()
        {
            Console.WriteLine("Domyślna powierzchnia figury: ");
            return 0;
        }
    }
    class Prostokat : Ksztalt
    {
        public Prostokat(int a, int b) : base(a, b)
        {
        }
        public override int Powierzchnia()
        {
            Console.WriteLine("Powierzchnia kwadratu: ");
            return (wysokosc * szerokosc);
        }
```

```
    }
    class Trojkat : Ksztalt
    {
        public Trojkat(int a, int b) : base(a, b)
        {
        }
        public override int Powierzchnia()
        {
            Console.WriteLine("Powierzchnia trójkąta: ");
            return (wysokosc * szerokosc) / 2;
        }
    }
    class BezImplementacji : Ksztalt
    {
        public BezImplementacji(int a, int b) : base(a, b)
        {
        }
    }
    // Co za konstrukcja?
    // Każda z klas pochodnych w programie dziedziczy po klasie bazowej
    // Posiada jedynie inne implementacje metody Powierzchnia()
    // W poniżej konstrukcji tworzymy klasę, która jako parametr przyjmuje klasę Ksztalt
    // Klasa Ksztalt - nasza klasa bazowa, która następnie wywołuje metodę Powierzchnia()
    // dla typu danych jaki został przekazany
    class WywolanieKlass
    {
        public void WywolajKlase(Ksztalt k)
        {
            int a;
            a = k.Powierzchnia();
            Console.WriteLine("Powierzchnia: {0}", a);
        }
    }
}
```

Zadania do zrobienia

1. Stwórz klasę **Licz** z:
   - publicznym polem **wartosc** przechowującym wartość liczbową.
   - metodą **Dodaj** przyjmującą jeden parametr i dodającą przekazaną wartość do wartości trzymanej w polu **wartosc**.
   - analogiczną operację **Odejmij**
   - w **Main** utwórz kilka obiektów klasy **Licz** i wykonaj różne operacje.

   a. Do klasy Licz dodaj **konstruktor** z jednym parametrem - który inicjuje pole **wartosc** na liczbę przekazaną w parametrze.
   b. Zmień widoczność pola na **private** i dodaj funkcję wypisującą stan obiektu (pole **wartosc**)

2. Stwórz klasę **Sumator** z:
   - publicznym polem **Liczby** będącym tablicą liczb
   - metodą **Suma** zwracającą sumę liczb z pola Liczby
   - metodę **SumaPodziel3** zwracającą sumę liczb z tablicy, które są podzielne przez 3

   a. Zmień widoczność pola **Liczby** na **private** oraz dodaj **konstruktor**.
   b. Dodaj metodę: int **IleElementów** () zwracającej liczbę elementów na w tablicy
   c. Dodaj metodę wypisującą wszystkie elementy tablicy
   d. Dodaj metodę przyjmującą dwa parametry: **lowIndex** oraz **highIndex**, która wypisze elementy o indeksach >= lowIndex oraz <= highIndex. Metoda powinna zadziałać poprawnie, gdy lowIndex lub highIndex wykraczają poza zakres tablicy (pominąć te elementy).

3. Napisz klasę **Address**, zawierającą wszystkie stosowne pola umożliwiające prowadzenie korespondencji. W klasie uwzględnij metody **Print**() i **Change**() (do wypisania i zmiany wartości pól).

4. Zaimplementuj hierarchię składającą się z następujących klas: **Person**, **Student** oraz **Teacher**. Poszczególne klasy zawierają następujące cechy:
   a. **Person**: imię, nazwisko, pesel, rok urodzenia oraz płeć
   b. **Student**: zawiera wszystkie cechy zawarte w klasie Person oraz numer legitymacji i klasę
   c. **Teacher**: zawiera wszystkie cechy zawarte w klasie Osoba oraz tytuł • (magister, inżynier, lub magister inżynier)
   d. Zbierz dane w 3 listach odpowiadającym odpowiednim klasom. Wyświetl zebrane dane.