Gajewski Andrzej

gajewskiandrzej@hotmail.com

Streszczenie

[Przyciągnij uwagę czytelnika interesującym streszczeniem. Zazwyczaj jest to krótkie podsumowanie dokumentu.   
Aby dodać zawartość, wystarczy kliknąć tutaj i zacząć wpisywanie.]

Programowanie w C#

Programowanie obiektowe

Spis treści

[Klasy 2](#_Toc525045560)

[Klasa a obiekt 2](#_Toc525045561)

[Budowa klasy 2](#_Toc525045562)

[Elementy klasy 2](#_Toc525045563)

[Konstruktory 2](#_Toc525045564)

[Tablice i listy obiektów 2](#_Toc525045565)

[Właściwości 2](#_Toc525045566)

[Składniki statyczne 2](#_Toc525045567)

[Struktury a klasy 2](#_Toc525045568)

[Dziedziczenie 2](#_Toc525045569)

[Dziedziczenie – wprowadzenie 2](#_Toc525045570)

[Dziedziczenie po klasie System.Object 2](#_Toc525045571)

[Klasy pochodne 2](#_Toc525045572)

[Metody wirtualne 2](#_Toc525045573)

[Klasy abstrakcyjne 2](#_Toc525045574)

[Interfejsy 2](#_Toc525045575)

[Interfejsy - wprowadzenie 2](#_Toc525045576)

[Interfejsy a kolekcje (listy) 2](#_Toc525045577)

[Implementacja interfejsu 2](#_Toc525045578)

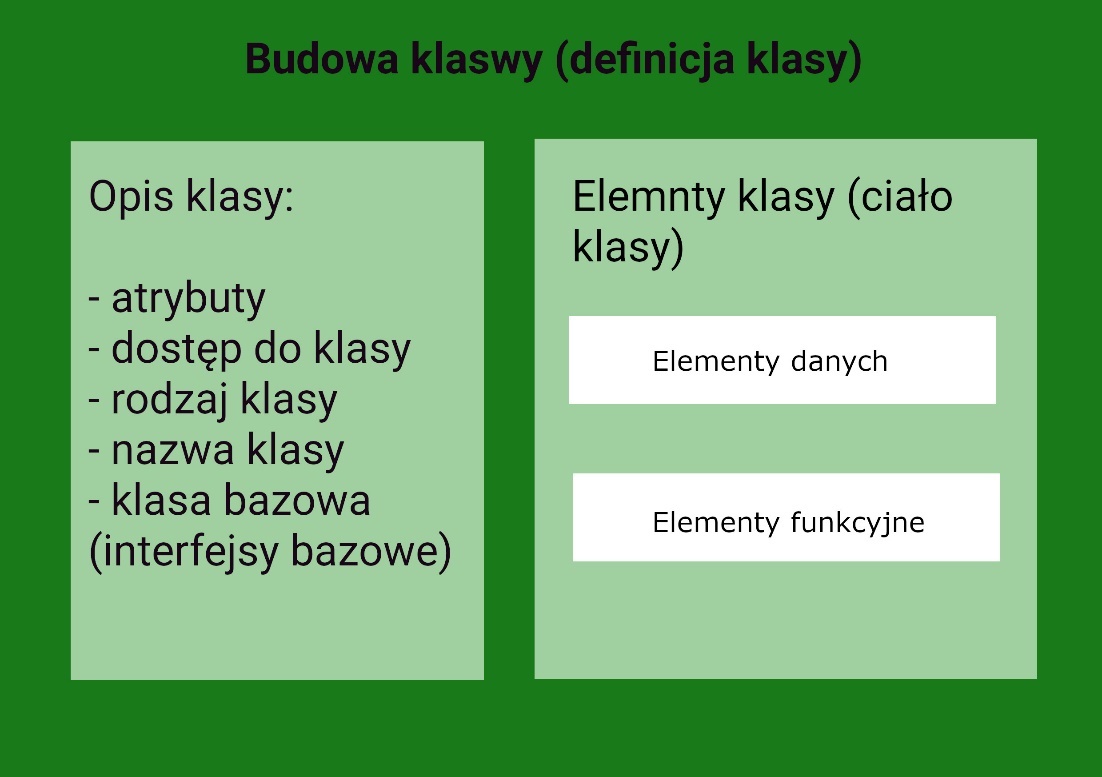
# Klasy

## Klasa a obiekt

**Klasa** to szablon określający, jak obiekt będzie wyglądał po utworzeniu jej instancji. Tak więc **obiekt** jest instancją klasy.

Nazwa klasy powinna opisywać obiekty, jakie będzie można na jej podstawie tworzyć. Zaleca się, aby nazwy klas zaczynały się z dużej litery, a jeśli nazwa ma klika słów, kolejne słowa także powinny zaczynać się z dużej litery, np. *PracownikAkordowy*Powyższe zalecenie odnoścnie nazw to tzw. Zasada **pascal case** i stosowana jest także dla nazw metod. Jest jeszcze inna zasada pisowni dla nazw, tzw. **camel case**, zgodnie z którą pierwsze słowo zaczyna się od małej litery, a kolejne od dużej, np., *wskazanieTeperatury*, stosowana jest zazwyczaj dla nazw zmiennych oraz argumentów.

## Budowa klasy



[Atrybuty][Modyfikatory] class Nazwa [:lista elementów bazowych]  
{  
 //Ciało klasy  
}

Modyfikatory dostępu: public, private, protected, internal, protected internal.

Modyfikatory określające rodzaj klasy:

* Abstract – modyfikator dla klasy abstrakcyjnej. Klasa abstrakcyjna wykorzystywana jest jedynie w postaci klasy bazowej i nie można na jej podstawie tworzyć obiektów
* Sealed – modyfikator dla klasy, która nie może pełnić roli klasy bazowej
* Static – modyfikator dla klasy statycznej, zawierającej wyłącznie składowe stayczne.

## Elementy klasy

Modyfikatory elementów klasy:

* Public – dany element klasy jest dostępny zarówno z wnętrza jak i spoza klasy
* Private – element jest dostępny tylko z wnętrza klasy – ten modyfikator jest domyślny dla elementów klasy
* Protected – daje dostęp do danego elementu z wnętrza klasy oraz klasom pochodnym
* Internal – dostęp dotyczy klas z tego samego zestawu
* Protected internal – składowa jest widoczna z wnętrza klasy, w której została zadeklarowana (lub klasy pochodnej od niej) oraz z wnętrza zestawu, w którym się znajduje)

Example:

namespace L12\_3\_ElementyKlasy

{

class Kwadrat

{

// Elementy danych:

int bok= 5;

ConsoleColor kolor= ConsoleColor.Green; // typ wyliczeniowy (enum) dla kolorów

// Elementy funkcyjne:

public void RysujKwadrat()

{

Console.ForegroundColor = kolor; // właściwość dla koloru znaków na konsoli

for (int i = 1; i <= bok; i++)

{

for (int j = 1; j <= bok; j++)

Console.Write(" \* ");

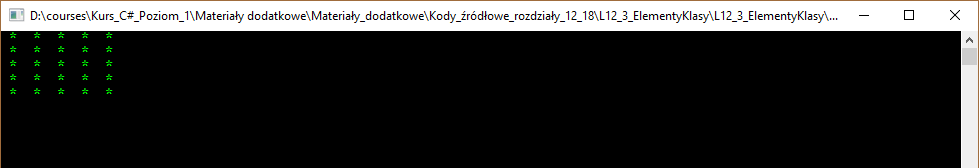
Console.WriteLine();

}

}

}

}



## Konstruktory

Example

namespace L12\_4\_Konstruktory

{

class Kwadrat

{

// Elementy danych:

int bok =4;

ConsoleColor kolor = ConsoleColor.Red; // typ wyliczeniowy (enum) dla kolorów

public static int liczbaKwadratow;

// Elementy funkcyjne:

public Kwadrat(int bok1, ConsoleColor kolor1) // Konstruktor

{

bok = bok1;

kolor = kolor1;

liczbaKwadratow++;

}

public Kwadrat()

{

liczbaKwadratow++;

}

static Kwadrat() //konstruktor statyczny

{

liczbaKwadratow = 0;

}

public Kwadrat(Kwadrat kopia) // konstruktor kopiujący

{

this.bok = kopia.bok;

this.kolor = kopia.kolor;

liczbaKwadratow++;

}

public void RysujKwadrat()

{

Console.ForegroundColor = kolor; // właściwość dla koloru znaków na konsoli

for (int i = 1; i <= bok; i++)

{

for (int j = 1; j <= bok; j++)

Console.Write(" \* ");

Console.WriteLine();

}

}

}

class Program

{

static void Main(string[] args)

{

Kwadrat k1 = new Kwadrat(5, ConsoleColor.Green); // Utworzenie obiektu

Kwadrat k2 = new Kwadrat();

k1.RysujKwadrat(); // Wywołanie metody rysującej kwadrat

k2.RysujKwadrat();

Kwadrat k3 = new Kwadrat(k2);

k3.RysujKwadrat();

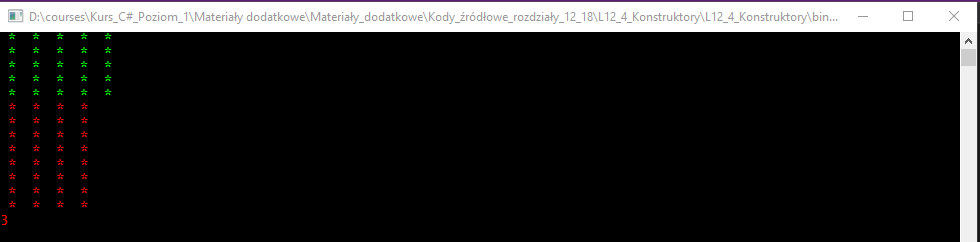
Console.WriteLine(Kwadrat.liczbaKwadratow);

Console.ReadKey();

}

}

}



## Tablice i listy obiektów

class Program

{

static void Main(string[] args)

{

// Kwadrat k1 = new Kwadrat(5, ConsoleColor.Green);

// Tablica obiektów:

//Kwadrat[] tab = new Kwadrat[3];

//tab[0] = new Kwadrat(3, ConsoleColor.White);

//tab[1] = new Kwadrat(4, ConsoleColor.Yellow);

//tab[2] = new Kwadrat(5, ConsoleColor.Blue);

Kwadrat[] tab = { new Kwadrat(3, ConsoleColor.White),

new Kwadrat(4, ConsoleColor.Yellow),

new Kwadrat(5, ConsoleColor.Blue) };

foreach (Kwadrat kw in tab)

{

kw.RysujKwadrat();

}

Console.ForegroundColor = ConsoleColor.Gray;

foreach (Kwadrat kw in tab)

{

Console.WriteLine(kw.InformacjeOKwadracie());

}

// Lista obiektów

List<Kwadrat> listaKwadratow = new List<Kwadrat>(tab);

listaKwadratow.Add(new Kwadrat(2, ConsoleColor.DarkGreen));

Console.WriteLine("Kwadraty z listy:");

foreach (Kwadrat kw in listaKwadratow)

{

Console.WriteLine(kw.InformacjeOKwadracie());

}

Console.ReadKey();

}

}

}

## Właściwości

## Składniki statyczne

## Struktury a klasy

# Dziedziczenie

## Dziedziczenie – wprowadzenie

## Dziedziczenie po klasie System.Object

## Klasy pochodne

Example:

namespace KlasyPochodne

{

class Kwadrat

{

protected int bok;

protected ConsoleColor kolor; //typ wyliczeniowy (enum)

public Kwadrat(int bok1, ConsoleColor kolor1) //Konstruktor

{

bok = bok1;

kolor = kolor1;

}

protected virtual int ObliczPole()

{

return bok \* bok;

}

public void RysujKwadrat()

{

Console.ForegroundColor = kolor; //właściwość dla koloru

for(int i = 1; i<=bok; i++)

{

for(int j = 1; j<=bok; j++)

Console.Write(" \* ");

Console.WriteLine();

}

}

public override string ToString()

{

return String.Format("Bok {0} Kolor {1} Pole {2}",

bok, kolor, ObliczPole());

}

}

class Prostokat : Kwadrat

{

int bokB;

public Prostokat(int b1, ConsoleColor k1, int b2): base(b1,k1)

{

bokB = b2;

}

protected override int ObliczPole()

{

return bok \* bokB;

}

public void RysujProstokat()

{

Console.ForegroundColor = kolor;

for(int i = 1; i <= bok; i++)

{

for(int j = 1; j <=bokB; j++)

Console.Write(" \* ");

Console.WriteLine();

}

}

public override string ToString()

{

return String.Format("Bok {0}", bokB) + base.ToString();

}

}

class Program

{

static void Main(string[] args)

{

Kwadrat k1 = new Kwadrat(4, ConsoleColor.Yellow);

Prostokat p1 = new Prostokat(2, ConsoleColor.Red, 3);

Console.WriteLine(k1.ToString());

Console.WriteLine(p1.ToString());

k1.RysujKwadrat();

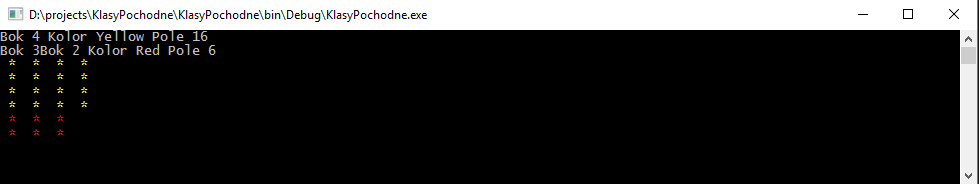
p1.RysujProstokat();

Console.ReadKey();

}

}

}



## Metody wirtualne

Mówimy w programowaniu o dwóch rodzajach poliformizmu: statycznym i dynamicznym. W wersji dynamicznej wielopostaciowość udostępniana jest poprzez metody wirtualne - konkretna wersja metody może być ustalona dopiero w czasie wykonywania programu (na podstawie typu obiektu). Jeżeli z jakiegoś powodu chcemy udaremnić poliformizm, możemy zastosować operator *new*, który pozwala zerwać więzy z klasą bazową.

Metody wirtualne nie mogą być prywatne ani statyczne!!!

Example:

namespace MetodyVirtualne

{

public class Pojazd //Klasa bazowa

{

public virtual void Jedzie()

{

Console.WriteLine("Pojazd jedzie");

}

}

public class Samochod : Pojazd

{

public override void Jedzie()

{

Console.WriteLine("Samochód jedzie");

}

}

public class Pociag : Pojazd

{

public override void Jedzie()

{

Console.WriteLine("Pociąg jedzie");

}

}

class Program

{

static void Main(string[] args)

{

List<Pojazd> listaPojazdow = new List<Pojazd>();

listaPojazdow.Add(new Pojazd());

listaPojazdow.Add(new Samochod());

listaPojazdow.Add(new Pociag());

foreach(Pojazd p in listaPojazdow)

{

p.Jedzie();

}

Console.ReadKey();

}

}

}

## 

## 

## Klasy abstrakcyjne

Klasa abstrakcyjna jest zdefiniowana jedynie na potrzeby dziedziczenia. Nie można utworzyć obiektu na jej podstawie.

Za pomocą słowa kluczowego abstract tworzy się metody abstrakcyjne. Taka metoda nie ma własnej implemantacji. Implementacją muszą się zająć klasy pochodne.

Metoda abstrakcyjna może być utworzona wyłącznie w klasie abstrakcyjnej.

Example:

namespace KlasyAbstracyjne

{

public abstract class Figura //Klasa abstrakcyjna

{

protected ConsoleColor kolor;

public Figura(ConsoleColor k1)

{

kolor = k1;

}

public override string ToString()

{

return String.Format("Kolor {0} Pole {1}", kolor, Pole());

}

public abstract double Pole(); //metoda abstrakcyjna

}

public class Prostokat : Figura

{

double bokA;

double bokB;

public Prostokat(ConsoleColor k1, double b1, double b2): base(k1)

{

bokA = b1;

bokB = b2;

}

public override double Pole()

{

return bokA \* bokB;

}

}

class Kolo : Figura

{

double promien;

public Kolo(ConsoleColor k1, double r1):base (k1)

{

promien = r1;

}

public override double Pole()

{

return Math.PI \* Math.Pow(promien, 2);

}

}

class Program

{

static void Main(string[] args)

{

List<Figura> listaFigur = new List<Figura>();

listaFigur.Add(new Prostokat(ConsoleColor.Red, 2, 4));

listaFigur.Add(new Kolo(ConsoleColor.Blue, 1));

foreach(Figura f1 in listaFigur)

{

Console.WriteLine(f1.ToString());

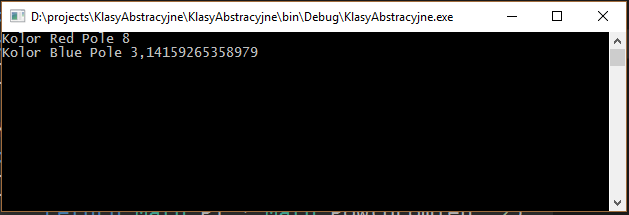
}

Console.ReadKey();

}

}

}



# Interfejsy

## Interfejsy – wprowadzenie

Interfejs definiuje klasę i jej elementy bez ich implementacji. Stanowi kontrakt nakazujący klasie określone działania. Klasa implementująca interfejs tworzy implementację wszystkich elementów zdefiniowanych w tym interfejsie. Definicja interfejsu przypomina definicję klasy. Zamiast słowa *class* jest **intrface**. Interfejs może zawierać jedynie metody (ich deklarację) i właściwości (oraz zdarzenia i indeksery). Ponadto przyjęto konwencję co do nazwy interfejsów – powinny zaczynać się od litery I, np. *IPracownik, IWektor*.

namespace Interfejsy

{

//Interfejs. Automatyczny dostęp dla interfejsu to public

interface IPunkt

{

//Składowymi interfejsu mogą być jedynie właściwości i metody

//nie podajemy dla nich modyfikatora dostępu

int x { get; set; } //właściwość

int y { get; set; } //właściwość

string DajOpis(); //deklaracja metody

}

class Punkt : IPunkt //Klasa Punkt implementuje interfejs IPunkt

{

public int x { get; set; } // Implementacja interfejsu-musi być public

public int y { get; set; } // Implementacja interfejsu-musi być public

public Punkt(int x1, int y1)

{

x = x1;

y = y1;

}

public string DajOpis() // Implementacja interfejsu-musi być public

{

return CosJeszcze() + String.Format("Współrzędne {0},{1}", x, y);

}

private string CosJeszcze() //Inna metoda w tej klasie

{

return "Opis: ";

}

}

class Program

{

static void Main(string[] args)

{

Punkt p1 = new Punkt(4, 6);

Console.Write(p1.DajOpis());

Console.ReadKey();

}

}

}

## Interfejsy a kolekcje (listy)

## Implementacja interfejsu