

# Programowanie obiektowe

## Wykład 3

Marcin Młotkowski

9 marca 2017

# Plan wykładu

- 1 Polimorfizm
  - Polimorfizm zawierania
  - Metody wirtualne
  - Pułapki dziedziczenia i metod wirtualnych
  - Programowanie rodzajowe (generyczne)
- 2 Polimorfizm ad-hoc — przeciążanie
  - Przeciążanie metod i operatorów
  - Overloading a overriding
- 3 Klasy jak obiekty
  - Pola i metody statyczne
  - Wzorzec projektowy Singleton
  - Inicjowanie klasy
  - Operatory

# Biblioteka figur geometrycznych

## Zadanie

implementacja biblioteki figur geometrycznych (Punkt, Kwadrat, Trójkąt).

# Klasa podstawowa

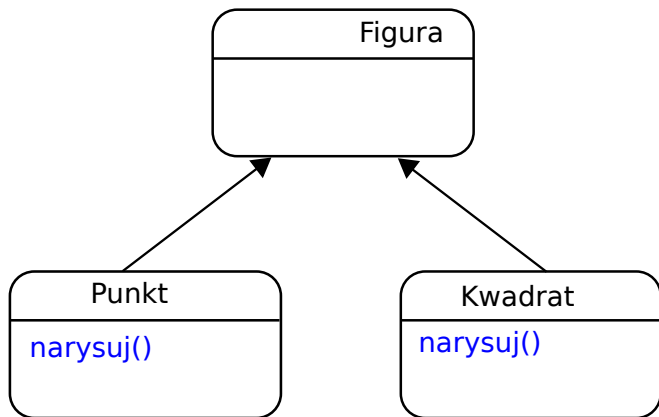
```
class Figura
{
    protected float x, y;
}
```

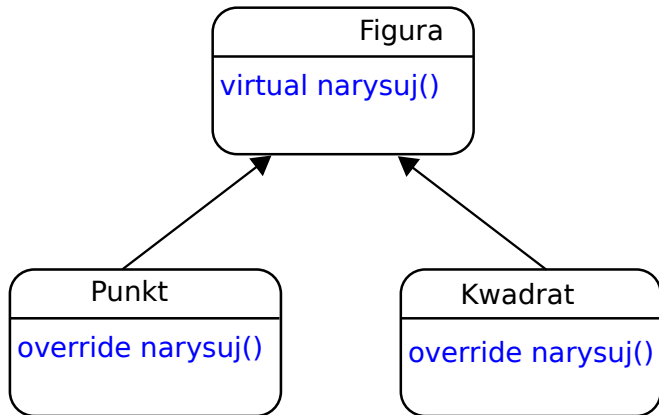
## Klasy pochodna

```
class Punkt : Figura
{
    public void narysuj(Color c)
    {
        ...
    }
}
```

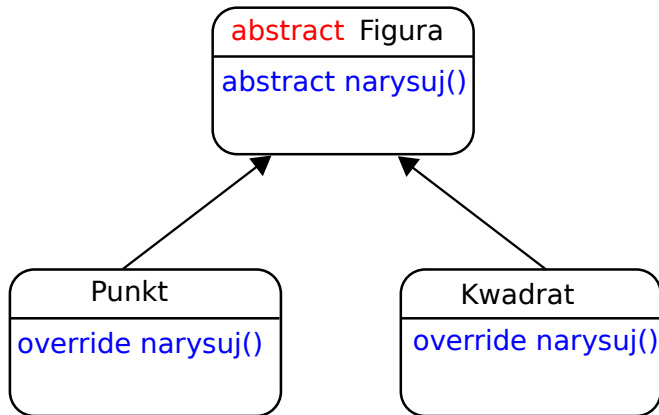
## Inna klasa pochodna

```
class Kwadrat : Figura
{
    public void narysuj(Color c)
    {
        ...
    }
}
```









# Deklaracja klasy abstrakcyjnej

```
abstract class Figura
{
    protected float x, y;
    public abstract void narysuj(Color c);
}
```

## Przykłady użycia

```
Figura[ ] obrazek = new Figura[3];  
obrazek[0] = new Punkt();  
obrazek[1] = new Kwadrat();  
obrazek[2] = new Punkt();  
  
foreach(Figura f in obrazek) f.narysuj();
```

## Inny przykład użycia

```
Figura p1 = new Kwadrat();
```

```
Figura p2 = new Punkt();
```

# Analiza przykładów

Figura p1 = new Kwadrat()



typ zmiennej



typ wartości

# Definicja

## Polimorfizm zawierania

Koncepcja, w której zmienna określonej klasy *C* może zawierać wartości — obiekty — klas będących podklasami *C*.

## Przykłady polimorfizmu zawierania

```
Figura p1 = new Kwadrat();
```

## Przykłady polimorfizmu zawierania

```
Figura p1 = new Kwadrat();
```

```
void foo(Figura p)
```

```
{
```

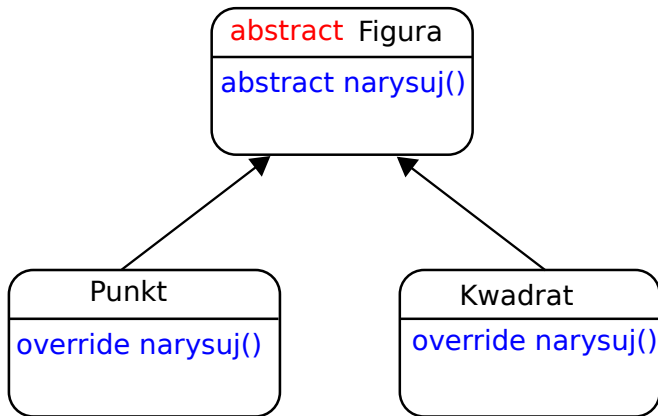
```
    ...
```

```
}
```

```
foo(new Kwadrat());
```



# Przypomnienie



# Przesuwanie figur

## Algorytm

- zmazanie starej zawartości, np. rysując w kolorze tła;
- zmiana współrzędnych;
- ponowne narysowanie figury, ale już w nowym położeniu.

## Implementacja algorytmu

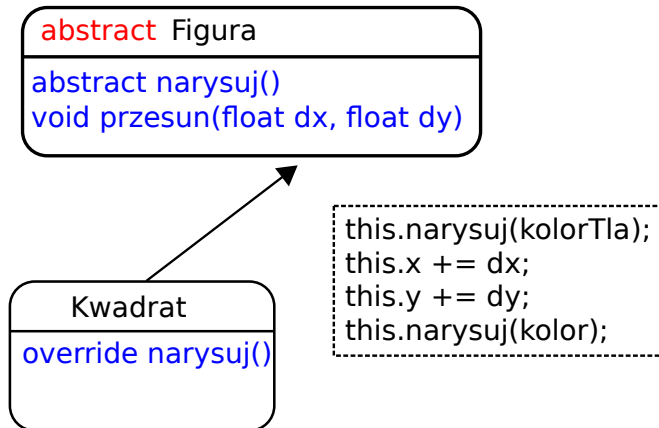
```
class Figura
public void przesun(float dx, float dy)
{
    this.narysuj(kolorTla);
    x += dx;
    y += dy;
    this.narysuj(kolor);
}
```

## Implementacja algorytmu

```
class Figura
public void przesun(float dx, float dy)
{
    this.narysuj(kolorTla);
    x += dx;
    y += dy;
    this.narysuj(kolor);
}
```

```
Kwadrat k = new Kwadrat();
k.przesun(1.234, -5.678);
```

## Analiza przykładu



# Wywołania metod wirtualnych

```
this.narysuj()
```

## Wywołania metod wirtualnych

```
this.narysuj()
```

```
Figura[ ] obrazek = new Figura[3];  
obrazek[0] = new Punkt();  
obrazek[1] = new Kwadrat();  
obrazek[2] = new Punkt();  
foreach(Figura f in obrazek) f.narysuj();
```

# Metody polimorficzne, uwagi

## Podsumowanie

- Pierwsza definicja metody musi być **abstract** lub **virtual** ;
- deklaracje metod w kolejnych klasach muszą być **override** ;
- ponownie zdefiniowane metody muszą mieć dokładnie te same parametry i ten sam zwracany typ.



# Metody niewirtualne

```
class Klasa
{
    public void foo() { Console.WriteLine("Klasa.foo()"); }
}
```

```
class Podklasa : Klasa
{
    new public void foo() {
        Console.WriteLine("Podklasa.foo()");
    }
}
```

# Metody niewirtualne

```
class Klasa
{
    public void foo() { Console.WriteLine("Klasa.foo()"); }
}
```

```
class Podklasa : Klasa
{
    new public void foo() {
        Console.WriteLine("Podklasa.foo()");
    }
}
```

## Przykład

```
Klasa k = new Podklasa();
k.foo();
```

# Metody niewirtualne

```
class Klasa
{
    public void foo() { Console.WriteLine("Klasa.foo()"); }
}
```

```
class Podklasa : Klasa
{
    new public void foo() {
        Console.WriteLine("Podklasa.foo()");
    }
}
```

## Przykład

```
Klasa k = new Podklasa();
k.foo();
```

"Klasa.foo()"

# Klasa Prostokąt

```
public class Prostokąt
{
    float szerokość, wysokość;
    public virtual void Szerokość(float w) { this.szerokość = w; }
    public virtual void Wysokość(float w) { this.wysokość = w; }
    public float Pole()
    {
        return this.szerokość * this.wysokość;
    }
}
```

## Klasa Kwadrat

```
public class Kwadrat : Prostokąt {  
    public override void Szerokość(float w) {  
        this.szerokość = w;  
        this.wysokość = w;  
    }  
    public override void Wysokość(float w) {  
        this.wysokość = w;  
        this.szerokość = w;  
    }  
}
```

# Zagadka

Czy ta implementacja jest poprawna?

# Kontrprzykład

```
void test(Prostokąt p)
{
    p.Szerokość(4);
    p.Wysokość(5);
    if (p.Pole() != 20) Alert();
}
```

## Kontrprzykład

```
void test(Prostokąt p)
{
    p.Szerokość(4);
    p.Wysokość(5);
    if (p.Pole() != 20) Alert();
}
```

```
test(new Kwadrat())
```



## Odwrotna implementacja

```
class Kwadrat {  
    float szerokość;  
    public void Szerokość(float w) {  
        this.szerokość = w;  
    }  
}  
  
class Prostokąt : Kwadrat {  
    float wysokość;  
    public void Wysokość(float w) {  
        this.wysokość = w;  
    }  
}
```

# Obliczanie pola

```
class Kwadrat {  
    public virtual float pole() {  
        return this.szerokosc * this.szerokosc;  
    }  
}  
  
class Prostokąt : Kwadrat {  
    public override float pole() {  
        return this.szerokość * this.wysokość;  
    }  
}
```

# Kontrprzykład

```
Prostokąt p = new Prostokąt();  
p.Wysokość(4.0);
```

# Kontrprzykład

```
Prostokąt p = new Prostokąt();  
p.Wysokość(4.0);  
foo(p);
```

# Kontrprzykład

```
void foo(Kwadrat k) {  
    k.Szerokość(5.0);  
    k.Pole() ?????  
}
```

```
Prostokąt p = new Prostokąt();  
p.Wysokość(4.0);  
foo(p);
```

# Reguła projektowa

## Zasada podstawienia Liskov

Klasy powinny być tak zaprojektowane, aby w dowolnym programie zastąpienie obiektów klasy bazowej obiektami podklasy nie zmieniało zachowania programu.

# Programowanie generyczne

## Programowanie obiektowe

modelowanie danych rzeczywistych

## Struktury danych

stosy, kolejki, drzewa binarne, kopce etc.

## Implementacja listy, 1. podejście

```
class List {  
    List next;  
    protected object val;  
    public void Add(object val) {  
        if (this.next != null) this.next.Add(val);  
        else {  
            this.next = new List();  
            this.next.val = val;  
        }  
    }  
    public object Top() { return this.val; }  
}
```



## Przykład użycia

```
List l = new List();  
l.Add(4);  
l.Add(8);  
Console.WriteLine(l.Top());
```

## Rozwiązanie generyczne

```
class Lista {  
    Lista next;  
    protected object val;  
    public void Add(object val) {  
        if (this.next != null) this.next.Add(val);  
        else {  
            this.next = new Lista();  
            this.next.val = val;  
        }  
    }  
    public object Top() { return this.val; }  
}
```

## Rozwiązanie generyczne

```
class Lista {  
    Lista next;  
    protected  val;  
    public void Add( val) {  
        if (this.next != null) this.next.Add(val);  
        else {  
            this.next = new Lista();  
            this.next.val = val;  
        }  
    }  
    public  Top() { return this.val; }  
}
```

## Rozwiązanie generyczne

```
class Lista<T> {  
    Lista<T> next;  
    protected T val;  
    public void Add(T val) {  
        if (this.next != null) this.next.Add(val);  
        else {  
            this.next = new Lista<T>();  
            this.next.val = val;  
        }  
    }  
    public T Top() { return this.val; }  
}
```

## Przykład użycia

```
Lista<int> l;  
l.Add(5);  
Console.WriteLine(l.Top());
```

## Inne przykłady

```
class Dictionary<TKey,TValue>
```

```
{
```

```
    ...
```

```
}
```

```
class KsiazkaTelefoniczna: Dictionary<string, int>
```

```
{
```

```
    ...
```

```
}
```

# Biblioteka standardowa

System.Collection.Generics

## Ograniczenia typu — motywacje

### Implementacja zbioru w formie np. listy

Aby uniknąć powtórzeń wymagamy, aby obiekty implementowały metodę porównywania obiektów, np. `CompareTo()`.

### Implementacja drzew przeszukiwań

Obiekty winne implementować metodę porównywania obiektów.



# Składnia

```
class Zbiór<T> where T : IComparable<T>  
{  
    ...  
}
```

## Co to jest IComparable<T>?

- klasa
- interfejs
- ...

# Plan wykładu

- 1 Polimorfizm
  - Polimorfizm zawierania
  - Metody wirtualne
  - Pułapki dziedziczenia i metod wirtualnych
  - Programowanie rodzajowe (generyczne)
- 2 Polimorfizm ad-hoc — przeciążanie
  - Przeciążanie metod i operatorów
  - Overloading a overriding
- 3 Klasy jak obiekty
  - Pola i metody statyczne
  - Wzorzec projektowy Singleton
  - Inicjowanie klasy
  - Operatory

# Przypomnienie I

## Liczby typu `int`

$$2 + 2$$

$$3 * 4$$

$$x * (y + z)$$

## Liczby typu `float`

$$2.71 + 3.0$$

$$2 * 3.14$$

$$x * (y + z)$$

## Przypomnienie I

### Liczby typu `int`

$$2 + 2$$

$$3 * 4$$

$$x * (y + z)$$

### Liczby typu `float`

$$2.71 + 3.0$$

$$2 * 3.14$$

$$x * (y + z)$$

### Macierze

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad x * (y + z)$$

## Przypomnienie II

```
class Pojazd {  
    string marka;  
    int rok_prod;  
    public Pojazd()  
    {  
        this.marka = "Syrena";  
        this.rok_prod = 2010;  
    }  
    public Pojazd(string marka)  
    {  
        this.marka = marka;  
    }  
}
```

## Definicja przeciążania

**Przeciążanie nazwy (polimorfizm ad-hoc):** występowanie wielu metod różniących się istotnie typami argumentów.

## Dalsze przykłady

### Przeciążone standardowe metody statyczne w C#

`Console.Write(bool)`

`Console.Write(int)`

`Console.Write(float)`

`Console.Write(object)`

# Overriding

```
class Pojazd {  
    public virtual void start() {  
        Console.WriteLine("Start");  
    }  
}
```

```
class Samochod : Pojazd {  
    public override void start() {  
        Console.WriteLine("Start");  
    }  
}
```



# Overloading

```
class Paliwo { }  
class Benzyna : Paliwo { }  
class Pojazd {  
    public void start(Paliwo p) {  
        Console.WriteLine("Start na paliwie");  
    }  
}  
class Samochod : Pojazd {  
    public void start(Benzyna b) {  
        Console.WriteLine("Start na benzynie");  
    }  
}
```

# Plan wykładu

- 1 Polimorfizm
  - Polimorfizm zawierania
  - Metody wirtualne
  - Pułapki dziedziczenia i metod wirtualnych
  - Programowanie rodzajowe (generyczne)
- 2 Polimorfizm ad-hoc — przeciążanie
  - Przeciążanie metod i operatorów
  - Overloading a overriding
- 3 Klasy jak obiekty
  - Pola i metody statyczne
  - Wzorzec projektowy Singleton
  - Inicjowanie klasy
  - Operatory

# Wstęp

Klasy czasem mogą przypominać obiekty

- klasy mają własne pola;
- klasy mają własne metody;
- klasy dziedziczą pola i metody po nadklasach;
- klasy mają nawet własne konstruktory.

# Pola i metody statyczne

## Właściwości

- są częścią klasy, istnieją od momentu deklaracji klasy;
- obiekty również mogą korzystać z metod i pól statycznych

# Składnia

## Przykład: zliczanie liczby obiektów

```
class Klasa {  
    static int Licznik = 0;  
    public Klasa() {  
        Licznik++;  
    }  
    public static void Info() {  
        Console.WriteLine("Liczba obiektów: {0}", Licznik);  
    }  
}
```

# Składnia

## Przykład: zliczanie liczby obiektów

```
class Klasa {  
    static int Licznik = 0;  
    public Klasa() {  
        Licznik++;  
    }  
    public static void Info() {  
        Console.WriteLine("Liczba obiektów: {0}", Licznik);  
    }  
}
```

## Zastosowanie

```
Klasa.info();
```

## Użyteczne funkcje i stałe

```
class Const
{
    public static float Pi = 3.1415;
    public static float e = 2.7182;
    public static float sin(float a)
    {
        return a - a*a*a/6 + a*a*a*a*a/120;
    }
}
```

## Kolejny przykład

Zmiennych globalnych nie ma.



# Zmienne globalne

## Symulowanie zmiennych globalnych

```
class Katalogi {  
    static public string Obrazki = "C:\\Documents and Settings\\";  
    static public string tmp = "/tmp";  
    static public string bin = "/usr/bin";  
}
```

## Ograniczenie liczby instancji

Czasem jest niepożądane, aby istniał więcej niż jeden obiekt danej klasy:

- obsługa kolejki do drukarki;
- obsługa połączenia z bazą danych;
- logowanie zdarzeń.

# 1. implementacja Singletonu

```
sealed class Singleton
{
    Singleton() {}
    static Singleton instance;
    public string nazwa;
    public static Singleton Instance()
    {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

# 1. implementacja Singletonu

```
sealed class Singleton
{
    Singleton() {}
    static Singleton instance;
    public string nazwa;
    public static Singleton Instance()
    {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

```
Singleton a = Singleton.Instance();
Singleton b = Singleton.Instance();
```

# Objaśnienie

## Uwagi do implementacji

- konstruktor klasy jest prywatny, widoczny tylko dla metod statycznych;
- klasa jest **sealed**, tj. nie ma możliwości zbudowania jej podklasy;
- jest to "leniwy" singleton, obiekt jest budowany dopiero przy pierwszym odwołaniu.

## 2. (gorliwa) implementacja singletonu

```
sealed class Singleton
{
    static readonly Singleton inst = new Singleton();
    Singleton() {}
    public static Singleton Instance()
    {
        return inst;
    }
}
```

# Konstruktory klasy

## Przykład

```
class Klasa {  
    static Klasa()  
    {  
        ...  
    }  
}
```

# Konstruktory klasy

## Przykład

```
class Klasa {  
    static Klasa()  
    {  
        ...  
    }  
}
```

## Wykonanie konstruktora

Konstruktor klasy jest wykonywany tylko raz, w momencie ładowania klasy.



## Ponowne definiowanie pól i metod statycznych

```
class FloatConsts
```

```
{  
    public static Pi = 3.1415;  
}
```

```
class DoubleConsts : FloatConsts
```

```
{  
    public static new Pi = 3.14159265358979323846;  
}
```

# Operatory

```
class Matrix  
{  
    float[ ] store;  
}
```

# Operatory

```
class Matrix  
{  
    float[ ] store;  
}
```

```
Matrix m1, m2, m3;  
...  
m3 = m1 + m2;
```

## Deklaracja własnych operatorów

```
class Matrix
{
    float[ ] store;
    public static Macierz operator+(Macierz t1, Macierz t2)
    {
        return new Macierz();
    }
}
```