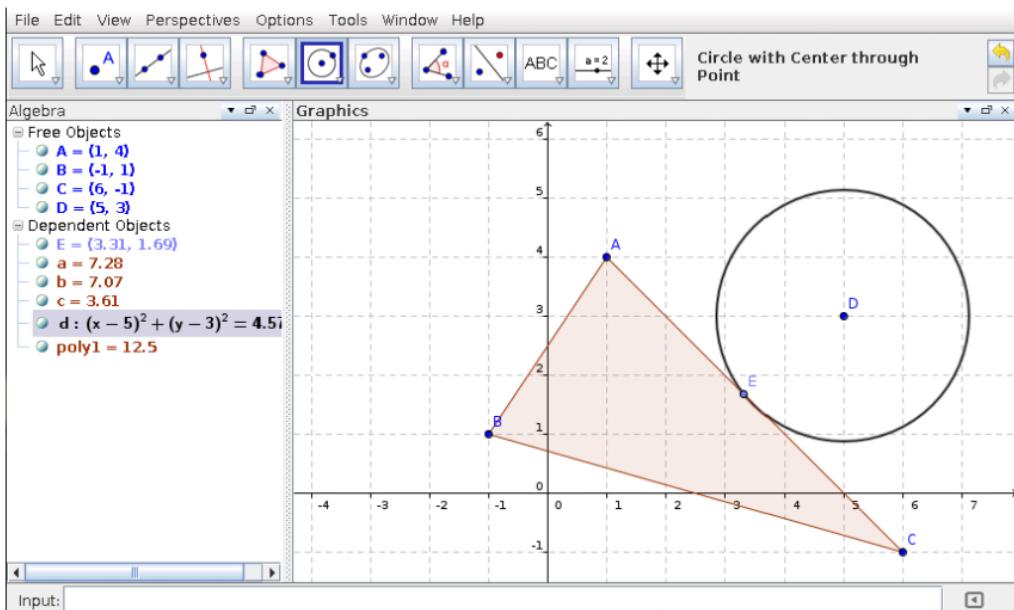


Lab. C#. Hierarchie klas. Figury

Planujesz stworzyć aplikację graficzną, w której - w układzie kartezjańskim - reprezentowane będą obiekty geometryczne: punkty, odcinki, okręgi, koła, trójkąty, ... Przykładem jest - pewnie znany Ci - program [Geogebra](#).



W zadaniu zauważamy dziedzinę problemu do 2-wymiarowego układu kartezjańskiego.

Wykonuj ćwiczenie krok po kroku.

Krok 1

- Utwórz Solution `FiguresLab`. Utwórz w nim 2 projekty:
 - projekt typu *Class Library* o nazwie `FiguresLib` - w którym zdefiniujesz hierarchię klas figur
 - projekt typu *Console Application* o nazwie `FiguresTestApp`, w którym testować będziesz rozwiązania tworzone w `FiguresLib`.
 - Powiąż referencyjnie projekt typu *console application* z projektem `FiguresLib`.
-

Krok 2

Kolejne kroki dotyczą projektu `FiguresLib`.

Klasa `Figure`

Utwórz abstrakcyjną klasę `Figure`. Reprezentować będzie ona każdą figurę geometryczną, rysowaną w Twojej aplikacji.

dokumentacja: Słowo kluczowe: `abstract`

```
namespace FiguresLib
{
    abstract public class Figure
    {
    }
}
```

- Każda figura ma etykię (typu `string`), modyfikowaną w dowolnym momencie. Etykieta służyć będzie do opisania obiektu na ekranie. Domyślną etykietą jest nazwa klasy obiektu oraz kolejny numer.

```
public string Label { get; set; }
```

- Figury rysowane będą określonym kolorem - domyślnie czarnym. W przestrzeni nazw `System.Drawing` zdefiniowana jest klasa `color` dostarczająca reprezentacje barw w systemie ARGB (alpha, red, green, blue).

Wykorzystamy ją.

```
public System.Drawing.Color Color { get; set; }
```

- Konstruktor domyślny **każdej** figury - ustawiamy domyślną wartość `Label` na nazwę klasy i kolejny unikalny numer (`counter`). Każda figura utworzona na bazie klasy `Figure` "otrzyma" unikalny numer (zapewniamy to prywatnym statycznym polem `counter` inkrementowanym każdorazowo przy uruchomieniu konstruktora).

```
private static int counter = 0;
public Figure()
{
    counter++;
    this.Color = Color.Black;
    Label = $"{GetType().Name} #{counter}";
}
```

Przy tworzeniu obiektu z klasy dziedziczącej z `Figure` uruchamiany jest automatycznie ten konstruktor.

- Ponieważ na tym etapie nie potrafisz rysować figur, zadeklaruj metodę abstrakcyjną `void Draw()`.

```
abstract public void Draw();
```

- Na zakończenie zdefiniujmy przeładowanie metody `ToString()`.

```
public override string ToString() => $"Figure: {Label}";
```

Kompletny kod klasy (pliku):

```
using System;
using System.Drawing;

namespace FiguresLib
{
    abstract public class Figure
    {
        public string Label { get; set; }
        private static int counter = 0;
        public Color Color { get; set; }

        // constructor
        public Figure()
        {
            counter++;
            this.Color = Color.Black;
            Label = $"{GetType().Name} #{counter}";
        }

        abstract public void Draw();

        public override string ToString() => $"Figure: {Label}";
    }
}
```

Klasa Point

W kolejnym kroku zdefiniuj klasę `Point` opisującą obiekt geometryczny *punkt* - o współrzędnych `x` oraz `y` typu `double`.

- Ponieważ rozdzielcość ekranu nie jest wysoka, przyjmujemy reprezentację współrzędnych z dokładnością do 4 cyfr po przecinku. Zamiast odwoływać się na sztywno do liczby `4` zdefiniujmy stałą w klasie `Figure` o nazwie `FRACTIONAL_DIGITS`, do której - w miarę potrzeby będziemy mogli się odwoływać.

Zmodyfikuj klasę `Figure`, dodając:

```
public const int FRACTIONAL_DIGITS = 4;
```

- Przymijmy, że obiekty typu `Point` są niezmienne. Domyślny punkt ma współrzędne `(0.0, 0.0)` i kolor niebieski. Zapewniamy reprezentację współrzędnych z zadaną dokładnością.

```
public class Point
{
    // fields
    public readonly double X, Y; // immutability

    // constructor
    public Point(double x = 0.0, double y = 0.0)
    {
        X = Math.Round(x, Figure.FRACTIONAL_DIGITS);
        Y = Math.Round(y, Figure.FRACTIONAL_DIGITS);
        Color = Color.Blue;
    }
}
```

• **POINT** dziedziczy z klasy `Figure`.

```
public class Point : Figure
```

Ponieważ klasa `Point` nie będzie abstrakcyjna (chcemy tworzyć obiekty), zatem musimy zaimplementować abstrakcyjną metodę `Draw`.

```
public override void Draw()
{
    Console.WriteLine("drawing: " + $"Point({X}, {Y}), {Color}");
}
```

- Punkty są niezmiennicze i reprezentowane z zadaną dokładnością. Ponieważ są obiektami, zatem dwa punkty `P1(0,0)` oraz `P2(0,0)` **nie są równe** (w sensie `==`) - są dwoma różnymi obiektami. Aby zapewnić "równość" obiektów w naszym znaczeniu (tzn. dwa punkty są równe, jeśli mają takie same odpowiednie współrzędne), musimy zaimplementować interfejs `IEquatable<Point>`.

```
public class Point : Figure, IEquatable<Point>
```

Wymusza to na nas szereg czyności (<https://docs.microsoft.com/en-us/dotnet/api/system.object.equals?view=netstandard-2.0>):

- implementację metody `Equals(Point p)`
- przeladowanie (`override`) metody `Equals(object o)`
- przeladowanie metody `GetHashCode()`
- dodatkowo - dla wygody, przedefiniowanie operatorów `==` oraz `!=`.

Ten fragment wykracza poza ramy kursu podstawowego.

```
#region implementation of IEquatable<Point>
// https://docs.microsoft.com/en-us/dotnet/api/system.object.equals?view=netstandard-2.0
public bool Equals(Point other) => (other is null) ? false : (this.X == other.X && this.Y == other.Y);
public override bool Equals(Object obj)
{
    if (!this.GetType().Equals(obj.GetType())) return false;

    return this.Equals((Point)obj);
}

public override int GetHashCode() => unchecked( ((int) X << 2) ^ (int) Y );

static public bool operator ==(Point p1, Point p2)
{
    if (p1 is null && p2 is null) return true;
    if (p1 is null && !(p2 is null)) return false;

    //p1 != null
    return p1.Equals(p2);
}
static public bool operator !=(Point p1, Point p2) => !(p1 == p2);
#endregion
```

- Dostarczamy przeladowanie metody `ToString()`.

```
public override string ToString() => $"Point({X},{Y})";
```

Kompletny kod klasy:

```
public class Point : Figure, IEquatable<Point>
{
    // fields
    public readonly double X, Y; // immutability

    // constructor
    public Point(double x = 0, double y = 0)
    {
        X = Math.Round(x, Figure.FRACTIONAL_DIGITS);
        Y = Math.Round(y, Figure.FRACTIONAL_DIGITS);
        Color = Color.Blue;
    }

    public override void Draw()
    {
        Console.WriteLine("drawing: " + $"Point({X}, {Y}), {Color}");
    }

    public override string ToString() => $"Point({X},{Y})";

    #region implementation of IEquatable<Point>
    // https://docs.microsoft.com/en-us/dotnet/api/system.object.equals?view=netstandard-2.0
    public bool Equals(Point other) => (other is null) ? false : (this.X == other.X && this.Y == other.Y);
    public override bool Equals(Object obj)
    {
        if (!this.GetType().Equals(obj.GetType())) return false;
    }
}
```

```

        if (this.Equals(obj))
            return this.Equals((Point)obj);
    }

    public override int GetHashCode() => unchecked( ((int) X << 2) ^ (int) Y );

    static public bool operator ==(Point p1, Point p2)
    {
        if (p1 is null && p2 is null) return true;
        if (p1 is null && !(p2 is null)) return false;

        //p1 != null
        return p1.Equals(p2);
    }
    static public bool operator !=(Point p1, Point p2) => !(p1 == p2);
#endregion
}

```

Testujemy klasę Point

W projekcie `FiguresTestApp`, w klasie `Program` i metodzie `Main()` umieszczamy kod i uruchamiamy:

```

static void Main(string[] args)
{
    Console.WriteLine("== default constructor ==");
    Point P0 = new Point();
    Console.WriteLine(P0);
    P0.Draw();

    Console.WriteLine("\n== new point (1,2) ==");
    Point P1 = new Point(1, 2);
    Console.WriteLine(P1);
    P1.Draw();

    Console.WriteLine("\n== another new point (1,2) ==");
    Point P2 = new Point(1, 2);
    P2.Color = System.Drawing.Color.Red; // bo konflikt nazw
    Console.WriteLine(P2);
    P2.Draw();

    Console.WriteLine("P1 equals P2: " + (P1==P2));

    Console.WriteLine("\n== another new point (1.00001, 2.000009) ==");
    Point P3 = new Point(1.00001, 2.000009);
    P3.Color = System.Drawing.Color.Yellow; // bo konflikt nazw
    Console.WriteLine(P3);
    P3.Draw();

    Console.WriteLine("P1 equals P3: " + (P1 == P3));
}

```

Wynik:

```

== default constructor ==
Figure: Point #1
drawing: Point(0, 0), Color [Blue]

== new point (1,2) ==
Figure: Point #2
drawing: Point(1, 2), Color [Blue]

== another new point (1,2) ==
Figure: Point #3
drawing: Point(1, 2), Color [Red]
P1 equals P2: True

== another new point (1.00001, 2.000009) ==
Figure: Point #4
drawing: Point(1, 2), Color [Yellow]
P1 equals P3: True

```

Klasa LineSegment

Zdefiniuj samodzielnie klasę `LineSegment` opisującą odcinek i spełniającą następujące założenia:

1. Odcinek reprezentowany jest przez dwa punkty
2. Odcinek domyślny, to odcinek zdefiniowany przez dwa punkty o współrzędnych `(0.0, 0.0)` - czyli odcinek o zerowej długości.
3. Oczywiście należy zdefiniować konstruktor tworzący odcinek na podstawie zadanych dwóch punktów.
4. Obiekty typu `LineSegment` są zmienne (mutables), zatem - po utworzeniu odcinka, można zmienić jego punkty.
5. Dla odcinka możemy obliczyć jego długość (właściwość `double Length {get;}`).
6. Dwa odcinki są **takie same**, jeśli punkty definiujące ich początek i koniec są **takie same** z dokładnością do

- v. Dwa odcinki są takie same, jeśli punkty dominujące ich początek i końca są takie same, z dodatkością do kolejności (tzn. Odcinek(P₁, P₂) == Odcinek(P₂, P₁)).

Aby zrealizować ostatni punkt, musisz zaimplementować `Equals`. Wzoruj się na klasie `Point`.

Rozwiązanie znajdziesz na końcu opracowania.

Interfejsy

Obiekty geometryczne mogą być mierzalne, tzn. możemy tym obiektom przyporządkowywać pewne miary liczbowe.

Punkt nie jest mierzalny. Odcinek jest mierzalny w przestrzeni 1D - można wyznaczyć jego długość. (Innym przykładem jest np. łuk, łamana, czy nawet okrąg - mówimy o długości okręgu).

Niektóre figury mogą być mierzalne w przestrzeni 2D. Miarą w przestrzeni 2D jest pole powierzchni. Mierzalną w 2D jest kwadrat, koło (ale nie okrąg), czy nawet sfera - rysowana w 3D.

Interfejsy są konstrukcjami programistycznymi definiującymi określone właściwości klas je implementujących.

Definiujemy interfejsy `IMeasurable1D` oraz `IMeasurable2D`. Figura mierzalna w 1D jest jednocześnie mierzalna w 2D.

```
public interface IMeasurable1D
{
    double Length {get; }
}

public interface IMeasurable2D : IMeasurable1D
{
    double Circumference { get; }
    double Surface { get; }
}
```

Zmodyfikuj nagłówek klasy `LineSegment` - dodaj implementacje interfejsu `IMeasurable1D`. Musisz zaimplementować właściwość `Length`.

Klasa Circle

Zdefiniuj okrąg jako klasę `circle`:

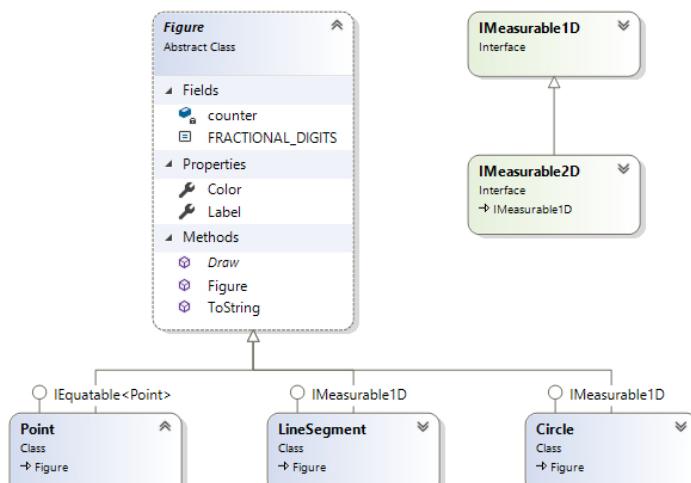
1. Okrąg jest przypadkiem figury (dziedziczy z `Figure`).
2. Okrąg zdefiniowany jest przez punkt, będący jego środkiem oraz długość promienia (`double`).
3. Okrąg domyślny: o środku w `(0, 0)` i promieniu `1`.
4. Okrąg jest *mutable*.
5. Okrąg jest `IMeasurable1D` (ale nie w 2D).
6. Zdefiniuj "równość" dwóch okręgów. Dwa okręgi są *takie same*, jeśli mają ten sam środek oraz promień o takiej samej długości.

Rozwiązanie znajdziesz na końcu opracowania.

Klasa Wheel

Zdefiniuj koło jako klasę `wheel`, bezpośrednio dziedziczącą z klasy `circle`. Obiekty tej klasy są mierzalne w 2D. Pamiętaj, konstruktory się nie dziedziczą.

Hierarchia klas i interfejsów:





Przetestuj funkcjonalność w [FiguresTestApp](#)

Zadania

Zadanie 1

Zdefiniuj strukturę `struct Vector` - w rozumieniu matematycznym: **wektor swobodny** (bez punktu zaczepienia).

Przykład: $\mathbf{a} = [1, 2]$, $\mathbf{b} = [-1, -1]$

Wektor nie jest figurą. Jest wykorzystywany do określania kierunku oraz przesunięć.

Wektor określony jest dwoma wartościami typu `double` - z dokładnością do `FRACTIONAL_DIGITS`: `x` oraz `y`.

Wektor jest *immutable*.

Wykorzystaj tę strukturę do przesuwania figur:

Zdefiniuj interfejs `IMoveable` z metodą

```
void Move( Vector v )
```

która przesuwa figurę o zadany wektor `v`.

Przymij, że `Point` nie jest przesuwalny, ale pozostałe figury już tak.

Zadanie 2

Dodaj do hierarchii klas klasę reprezentującą trójkąt (`Triangle`). Trójkąt zadany jest trzema punktami. Trójkąt jest mierzalny w 2D. Dodatkowo, możesz zaimplementować metody informujące o tym, czy trójkąt jest prostokątny, ostrokątny, rozwartokątny, równoboczny, równoramienny, ... Pewnie będziesz musiał przypomnieć sobie trochę geometrii analitycznej.

Zadanie 3

Dodaj do hierarchii klas klasę `Square`. Kwadrat wyznaczony jest przez dwa punkty, będące krańcami przekątnej. Przyda się umiejętność wyznaczania prostych prostopadłych.

Podsumowanie

Na tym prostym przykładzie zobaczyłeś, jak budowana jest architektura aplikacji, wykorzystując mechanizmy koncepcje klas struktur i interfejsów, dziedziczenia, hermetyzacji i polimorfizmu.

Rozwiązańia

Klasa `LineSegment` - listing

Listing - klasa `LineSegment`

```

public class LineSegment : Figure, IMeasurable1D, IEquatable<LineSegment>
{
    public Point StartPoint;
    public Point EndPoint;

    public LineSegment() : this( new Point(0, 0), new Point(0, 0) )
    {

```

```

        }

        public LineSegment( Point p1 , Point p2 )
        {
            StartPoint = p1;
            EndPoint = p2;
            Color = System.Drawing.Color.Green;
        }

        public double Length => Math.Sqrt( Math.Pow( StartPoint.X - EndPoint.X, 2) + Math.Pow( StartPoint.Y - EndPoint.Y, 2) );
    }

    public override void Draw()
    {
        Console.WriteLine($"drawing: {this}, {Color}, Length = {Length}");
    }

    public override string ToString() => $"LineSegment({StartPoint}, {EndPoint})";

#region IEquatable<LineSegment> implementation
    public bool Equals(LineSegment other)
    {
        if (other is null) return false;

        return (this.StartPoint == other.StartPoint && this.EndPoint == other.EndPoint)
            || (this.StartPoint == other.EndPoint && this.EndPoint == other.StartPoint);
    }

    public static bool operator ==(LineSegment s1, LineSegment s2)
    {
        if (s1 is null && s2 is null) return true;
        if (s1 is null && !(s2 is null) ) return false;

        return s1.Equals(s2);
    }

    public static bool operator !=(LineSegment s1, LineSegment s2) => !(s1 == s2);
#endregion
}

```

Listing - kod testujący

```

static void Main(string[] args)
{
    LineSegment s1 = new LineSegment();
    Console.WriteLine(s1);
    s1.Draw();

    LineSegment s2 = new LineSegment(new Point(1, 2), new Point(2, 3));
    s2.Draw();
    LineSegment s3 = new LineSegment(new Point(1, 2), new Point(2, 3));
    s3.Draw();
    LineSegment s4 = new LineSegment(new Point(2, 3), new Point(1, 2));
    s4.Draw();
    Console.WriteLine(s2 == s3);
    Console.WriteLine(s2 == s4);
}

```

Klasa Circle - listing

```

public class Circle : Figure, IMeasurable1D, IEquatable<Circle>
{
    public Point Center;
    private double radius;
    public double Radius
    {
        get => radius;
        set
        {
            if (value < 0)
                throw new ArgumentException("negative argument not allowed");
            radius = Math.Round(value, Figure.FRACTIONAL_DIGITS);
        }
    }

    public Circle() : this( new Point(0,0), 1) { }

    public Circle(Point center, double radius)
    {
        Center = center;
        Radius = radius;
        Color = System.Drawing.Color.Cyan;
    }

    public double Length => 2 * Math.PI * radius;
}

```

```
public override string ToString() => $"Circle( {Center}, {radius} )";

public override void Draw()
{
    Console.WriteLine($"drawing: {this}, {Color}, Length = {Length}");
}

#region IEquatable<Circle> implementation
public bool Equals(Circle other)
{
    if (other is null) return false;

    return this.Center == other.Center && this.Radius == other.Radius;
}
#endregion
}
```

Klasa Wheel - listing

```
public class Wheel : Circle, IMeasurable2D
{
    public Wheel() : this(new Point(0, 0), 1) { }

    public Wheel(Point center, double radius) : base(center, radius)
    {
        Color = System.Drawing.Color.Magenta;
    }

    public double Circumference => Length;

    public double Surface => Math.PI * Radius * Radius;

    public override string ToString() => $"Wheel( {Center}, {Radius} )";
    public override void Draw()
    {
        Console.WriteLine($"drawing: {this}, {Color}, Circumference = {Circumference}, Surface = {Surface}");
    }
}
```