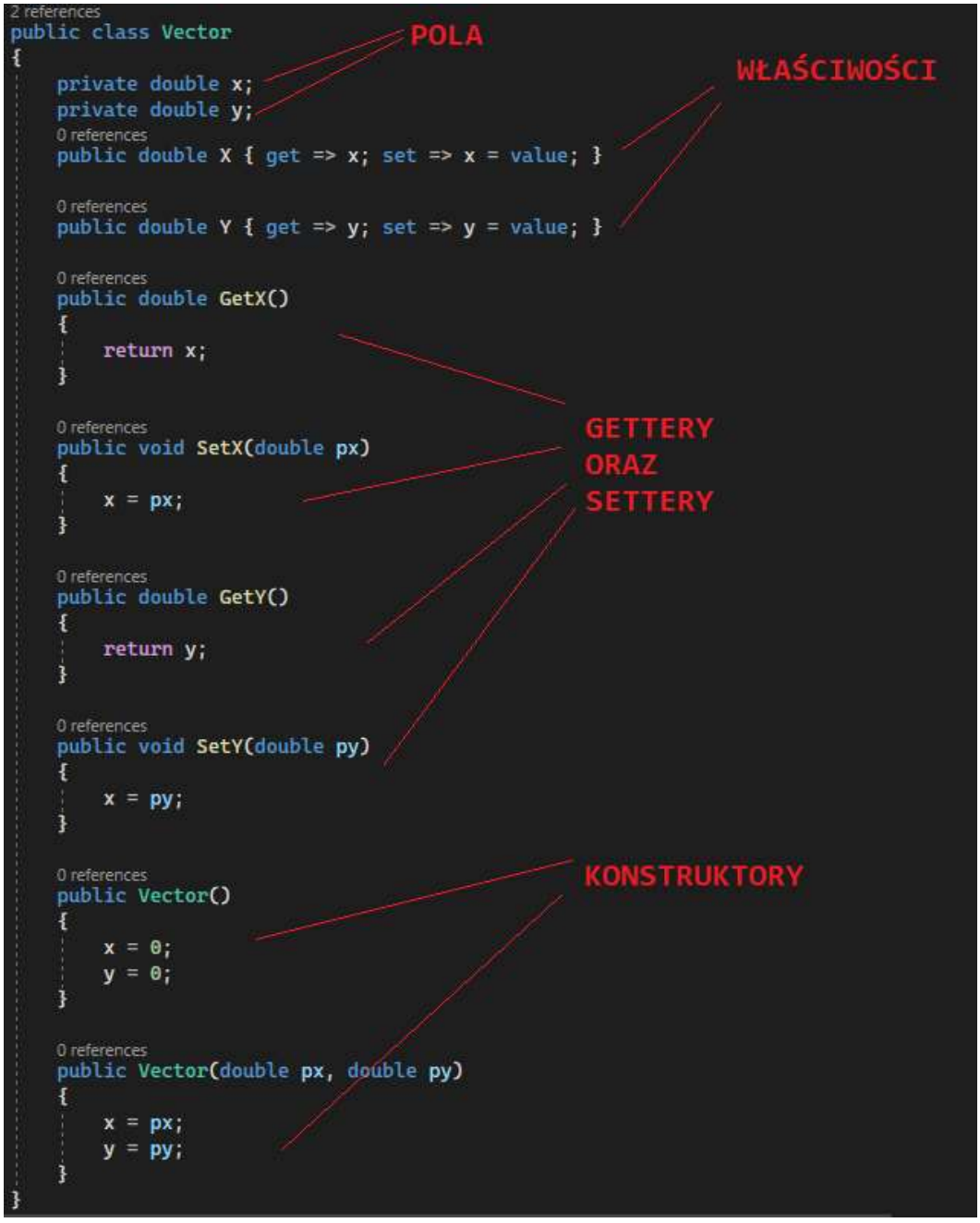
Zawartość:

- programowanie obiektowe w C#
    * pola i właściwości
    * metody
    * hermetyzacja
    * polimorfizm
    * klasy abstrakcyjne i interfejsy
    * elementy statyczne

PODSTAWOWY OOP W C#



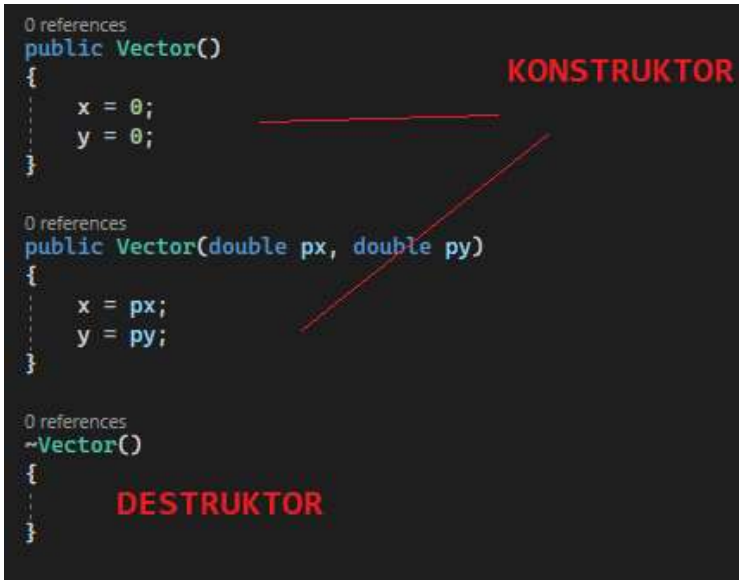

```
2 references
public class Vector
{
    private double x;
    private double y;
    0 references
    public double X { get => x; set => x = value; }

    0 references
    public double Y { get => y; set => y = value; }

    0 references
    public double GetX()
    {
        return x;
    }

    0 references
    public void SetX(double px)
    {
        x = px;
    }

    0 references
    public double GetY()
    {
        return y;
    }

    0 references
    public void SetY(double py)
    {
        x = py;
    }

    0 references
    public Vector()
    {
        x = 0;
        y = 0;
    }

    0 references
    public Vector(double px, double py)
    {
        x = px;
        y = py;
    }
}
```

Pole - zmienna w klasie, zachowuje się ona jak globalna zmienna w danej klasie.

Właściwość - element specyficzny dla języka C#, którego zadaniem jest ustawianie i oddawanie na zewnątrz prywatnych pól. Jest to najprościej mówiąc skrótowa wersja zapisu gettera i settera. Oczywiście lambdy umieszczone we właściwościach mogą być bardziej rozbudowane.

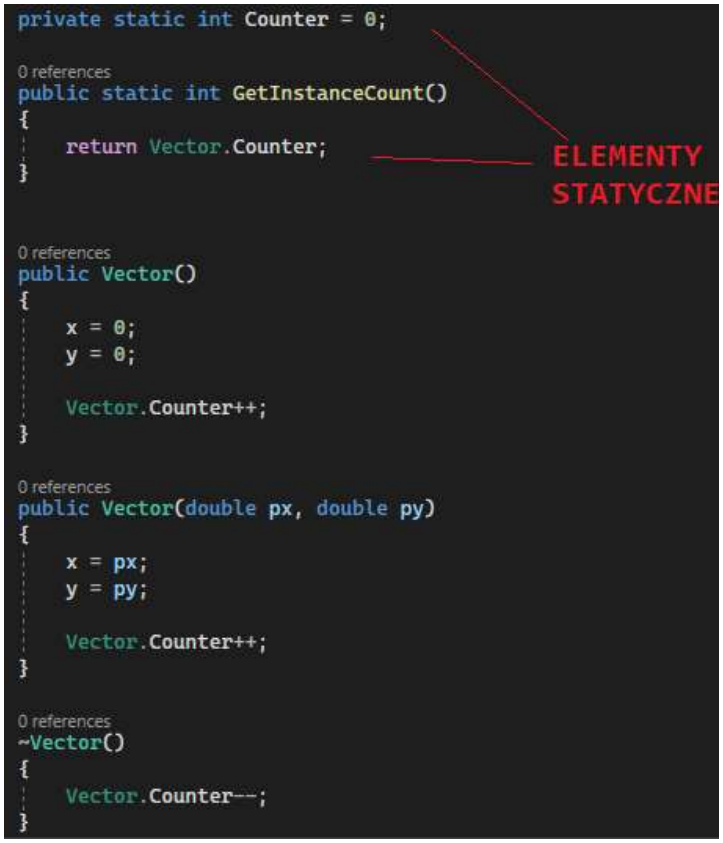
Gettery i Settery - metody służące do wystawiania na zewnątrz prywatnych pól (zasada hermetyzacji). To samo osiągniemy za pomocą właściwości.

Jak widać każdy element oraz cała klasa ma zdefiniowany stopień dostępności (public, protected, private, internal (ten modyfikator daje dostęp do elementu tylko w danym projekcie)). Ważne jest tutaj to, że elementy klasy nie mogą mieć większej dostępności niż cała klasa. Nie może dojść do sytuacji gdzie klasa jest prywatna albo internal a metody i pola są publiczne.

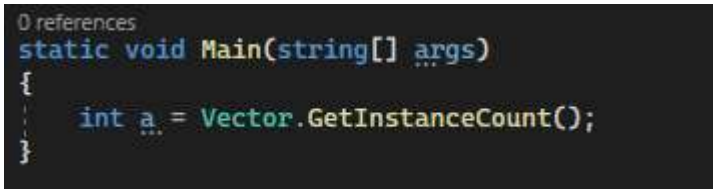


Konstruktor - metoda, której nazwa jest taka sama jak nazwa klasy, wykonywana w momencie tworzenia obiektu operatorem “new”. Jak widać konstruktory można przeciążać. Nie mają one także typu zwracanego.

Destruktor - metoda wywoływana w momencie niszczenia obiektu. W C# nie mamy kontroli kiedy następuje niszczenie obiektu. Zajmuje się tym mechanizm zwany Garbage Collector. W destruktorze powinniśmy zawrzeć logikę, która “sprząta” po obiekcie. Przykładem takiej akcji może być zamknięcie otwartych plików czy połączeń sieciowych bo GC usuwa tylko obiekt z pamięci.



Elementy statyczne to elementy, które są przypisane do całej klasy a nie do pojedynczego obiektu. Innymi słowy ujmując są one wspólne dla wszystkich obiektów danej klasy.
W powyższym przykładzie tworząc obiekt zwiększamy licznik obiektów danej klasy. A gdy GC go usunie licznik ten jest zmniejszany. Ten prosty przykład pozwala zliczać ilość stworzonych obiektów danej klasy.



Do elementów statycznych odwołujemy się za pomocą składnii nazwa_klasy.element_statyczny.

```
private double x;
private double y;
0 references
public double X { get => x; set => x = value; }

0 references
public double Y { get => y; set => y = value; }

0 references
public double GetX()
{
    return x;
}

0 references
public void SetX(double px)
{
    x = px;
}

0 references
public double GetY()
{
    return y;
}

0 references
public void SetY(double py)
{
    x = py;
}
```

Na powyższym screenie widać zastosowanie zasady hermetyzacji. Zasada ta, mówi by na zewnątrz klasy udostępniać minimalny wymagany do działania zestaw elementów. Stąd też kilka elementów na screenie jest prywatne.

Oczywiście stosując hermetyzacje nie musimy ukrywać elementów w pełni. Możemy utworzyć tak zwane gettery i settery (stare podejście znane z innych języków) lub właściwości, które pełnią rolę mostu między prywatnym elementem klasy a światem poza klasą. Mogą one zawierać dodatkową logikę, która przetwarza prywatne dane przed udostępnieniem ich na zewnątrz.

```
6 references
class Point
{
    protected int x;
    protected int y;

    0 references
    public Point()
    {

    }

    2 references
    public Point(int xp, int yp)
    {
        x = xp;
        y = yp;
    }

    2 references
    public virtual void PrintMe()
    {
        Console.WriteLine($"[{x};{y}]");
    }
}
```

```
2 references
class Point2D : Point
{
    0 references
    public Point2D()
    {

    }

    0 references
    public Point2D(int xp, int yp) : base(xp, yp)
    {

    }

    1 reference
    public override void PrintMe()
    {
        Console
            .WriteLine($"2D: [{base.x};{base.y}]");
    }
}
```

```
2 references
class Point3D : Point
{
    private int z;

    0 references
    public Point3D()
    {

    }

    0 references
    public Point3D(int xp, int yp, int zp)
        : base(xp, yp)
    {
        z = zp;
    }

    1 reference
    public override void PrintMe()
    {
        Console
            .WriteLine($"3D: [{base.x};{base.y};{z}]");
    }
}
```

Powyższy zrzut ekranu pokazuje prosty mechanizm dziedziczenia. Po lewej stronie zdefiniowano klasę bazową punkt. Posiada ona dwa pola, konstruktory oraz jedną metodę.

Punkt 2D dziedziczy z klasy Punkt, nie dodając od siebie praktycznie nic. Co warto zauważyć to sposób wywoływania konstruktora klasy bazowej (base()).

Punkt3D dodaje do tego co otrzymał z klasy Punkt jeszcze trzecią składową Z. Składowa ta jest ustawiana w konstruktorze. A pozostałe dwie są przekazywane do konstruktora klasy bazowej.

Jeżeli chcemy odwołać się do elementu klasy bazowej w klasie potomnej używamy konstrukcji:

base.element;

Oczywiście musi ona być publiczna lub protected.

Na powyższym zrzucie ekranu zastosowano jeszcze jedną technikę programowania obiektowego. Mowa o nadpisywaniu metod oraz wielopostaciowość.

Klasy pochodne nadpisują implementację PrintMe() uzyskaną z klasy bazowej.

```
Point p = new Point(2, 3);
Point2D p2 = new Point2D(2, 3);
Point3D p3 = new Point3D(2, 3, 5);

p.PrintMe();
p2.PrintMe();
p3.PrintMe();
```

```
[2;3]
2D: [2;3]
3D: [2;3;5]
```

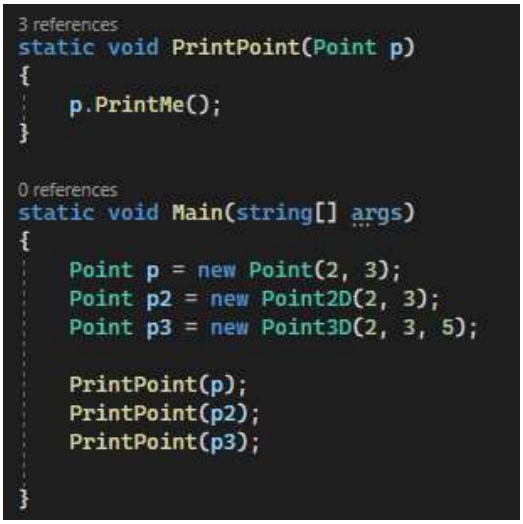
Prosty efekt napisania metody dziedziczonej własną implementacją. Sytuacja komplikuje się gdy zaczniemy korzystać z mechanizmu polimorfizmu. Przykład poniżej:

```
Point p = new Point(2, 3);
Point p2 = new Point2D(2, 3);
Point p3 = new Point3D(2, 3, 5);

p.PrintMe();
p2.PrintMe();
p3.PrintMe();
```

```
[2;3]
2D: [2;3]
3D: [2;3;5]
```

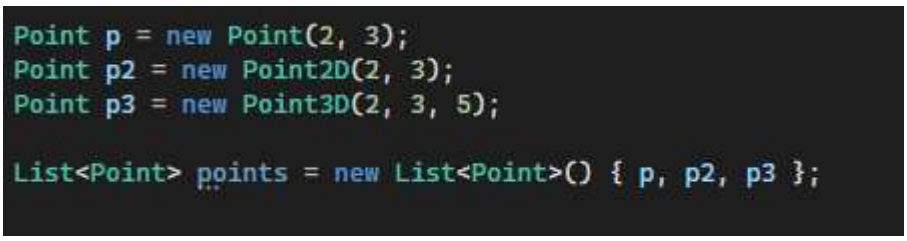
Widzimy tutaj przypisanie obiektu klasy pochodnej do zmiennej (referencji) typu bazowego. Zagrywka taka jest możliwa dzięki dziedziczeniu.

Inna sprawa to to co się wypisuje na ekranie. Ważną rolę odgrywa tutaj słowo kluczowe “virtual”. Sugeruje ono kompilatorowi, że w przypadku wywołania tej metody ma wykonać wersję przypisaną do typu obiektu a nie typu zmiennej (ujmując to prościej ma wykonać wersję przypisaną do typu po prawej stronie operatora “new”).



Dzięki polimorfizmowi i dziedziczeniu możliwa jest taka kombinacja jak na obrazku powyżej. Przekazujemy do funkcji zmienną typu Point, ale wykonanie PrintMe() i tak jest wykonywane na bazie typu użytego przy operatorze new. Dlatego za każdym razem wykona się metoda odpowiedniej klasy.

Usunięcie słowa kluczowego “virtual” usuwa ten efekt.



Kolejny przykład wielopostaciowości. Jeżeli wykonamy pętle po powyższej liście i na każdym uzyskanym obiekcie wykonamy PrintMe() za każdym razem wykona się odpowiednia metoda.

Klasy abstrakcyjne i interfejsy

```
1 reference
abstract class Figura
{
    protected double pole;
    protected double obwod;

    1 reference
    public abstract void ObliczPole();
    1 reference
    public abstract void ObliczObwod();

    0 references
    public virtual void Wyswietl()
    {
        System.Console.WriteLine($"Pole: {pole}, Obwód: {obwod}");
    }
}
```

```
1 reference
class Kwadrat : Figura
{
    private double a;

    0 references
    public Kwadrat(double a)
    {
        this.a = a;
    }

    1 reference
    public override void ObliczObwod()
    {
        obwod = 4 * a;
    }

    1 reference
    public override void ObliczPole()
    {
        pole = a * a;
    }
}
```

Klasa abstrakcyjna to klasa, która posiada metody bez implementacji. Klasy i metody takie oznacza się słowem kluczowym “abstract”.

```
1 reference
interface IFigura
{
    1 reference
    void ObliczPole();
    1 reference
    void ObliczObwod();
}

1 reference
class Kwadrat : IFigura
{
    private double a;
    private double pole;
    private double obwod;

    0 references
    public Kwadrat(double a)
    {
        this.a = a;
    }

    1 reference
    public void ObliczObwod()
    {
        obwod = 4 * a;
    }

    1 reference
    public void ObliczPole()
    {
        pole = a * a;
    }
    0 references
    public virtual void Wyswietl()
    {
        System.Console.WriteLine($"Pole: {pole}, Obwód: {obwod}");
    }
}
```

Interfejsy to specjalna konstrukcja języka C#. Jej działanie jest podobne do klas abstrakcyjnych. Jednak interfejsy nie mogą posiadać żadnej zaimplementowanej metody. Nie mogą także posiadać pól.

Interfejsy przyjęło się nazywać w C# zaczynając od dużej litery I.

Po co są właściwie te klasy abstrakcyjne i interfejsy?
Elementy te służą do wymuszania na osobie implementującej klasę potomną by ta zaimplementowała elementy oznaczone jako abstrakcyjne lub te wymienione w interfejsie. Klasę abstrakcyjną oraz interfejsy często nazywa się mianem “umów polimorficznych”. Dziedzicząc z interfejsu lub klasy abstrakcyjnej zobowiązujemy się zaimplementować składowe tam wymienione.

Sama idea interfejsów wzięła się z tego, że w C# można dziedziczyć z jednej klasy. Jeżeli chodzi o interfejsy to możemy dziedziczyć z dowolnej ilości.

```
0 references
static void Main(string[] args)
{
    IFigura figura = new Kwadrat(5);
}
```