Zaawansowane programowanie obiektowe - wykład 1

dr Piotr Jastrzębski

Sprawy ogranizacyjne

Sprawy ogranizacyjne

- Sylabus jest dostępny w systemie USOS.
- Forma zaliczenia: zaliczenie na ocenę.

{Ostatnia aktualizacja pliku: 2020-02-28 20:54:50.}

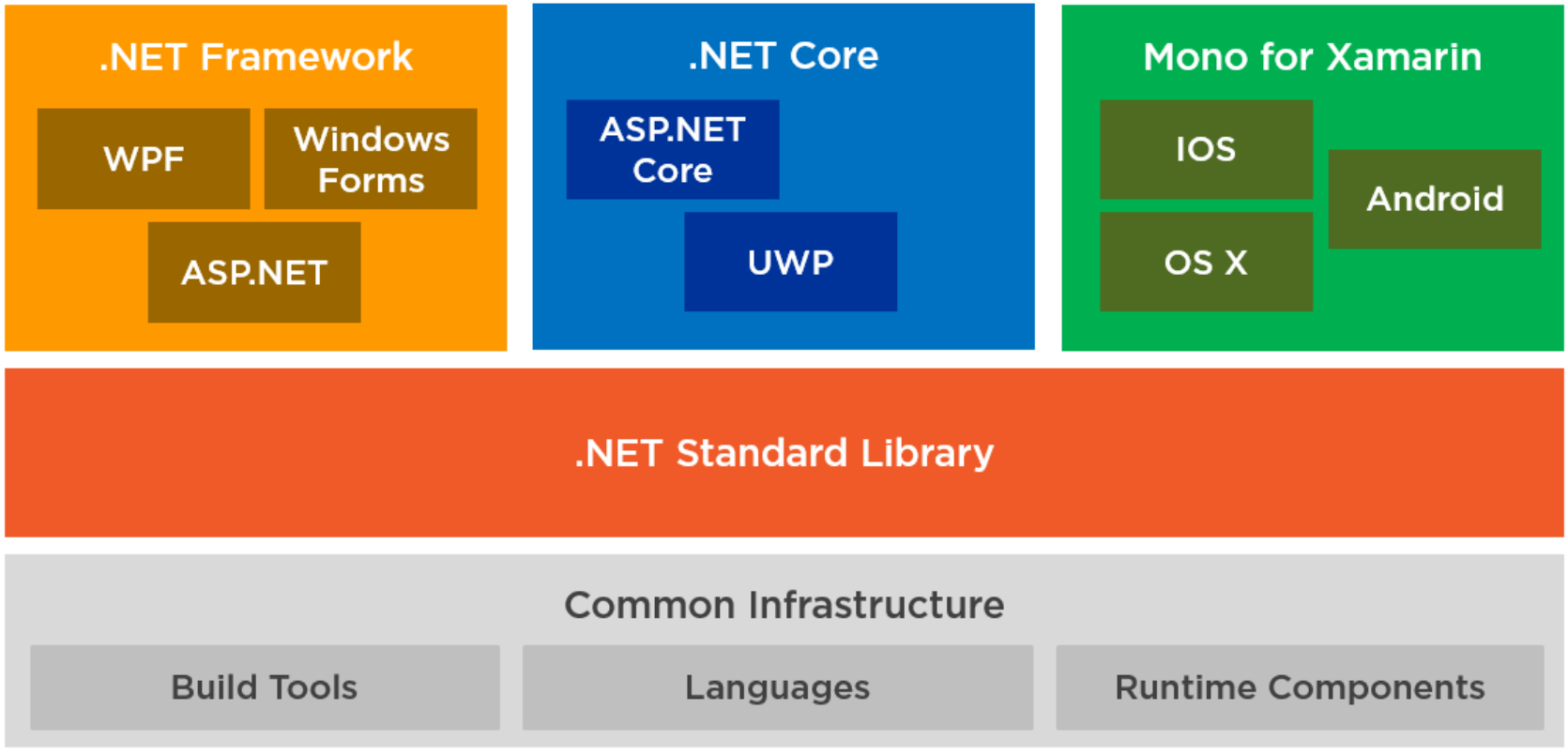
Wymagania wstępne

- Znajomość programowania strukturalnego (pętle, instrukcje warunkowe, funkcje, tablice, typy zmiennych, operacje arytmetyczne).
- Podstawy programowania obiektowego.
- Sprawna praca z Visual Studio.
- Podstawy debugowania kodu.

Język C#

C#

- Język C# powstał w roku 2000.
- Został stworzony przez Microsoft i jego zespół programistów pod kierunkiem Andersa Hejlsberga.
- Jest częścią ekosystemu .NET.



Rysunek 1: Ekosystem .NET

{Źródło zdjęcia: https://www.azurebarry.com/content/images/2017/08/Ecosystem.png.}

Programowanie obiektowe

Paradygmat obiektowy - po co jest?

- Jeśli chcemy zbudować 5 identycznych domów, nie potrzeba nam 5 projektów.
- Jeśli chcemy wyprodukować 4 samochody, nie potrzebujemy oddzielnych projektów.
- Programowanie obiektowe stara się pogrupować podobne „byty” na pewnym poziomie abstrakcji.

Programowanie strukturalne a obiektowe

- Programowanie strukturalne/proceduralne dzieli kod na części, moduły, wykorzystuje sekwencje, iteracje, funkcje.
- Programowanie obiektowe wprowadza pojęcie obiektu, w którym dane i procedury są ze sobą ściśle powiązane.

Klasa, obiekt

Przykład

- Chcemy napisać program wyliczający wartość rynkową domu w zależności od ilości pokoi, posiadania garażu, ogrodu, ilości pięter, powierzchni, itp. . .

Pierwsza klasa - `House`

- Na pewnym poziomie abstrakcji opisuje kawałek rzeczywistości.
- Zawiera dane opisujące możliwe stany.
- Zawiera metody opisujące możliwe zachowania.
- Możemy ją zastosować do wielu budynków.

```
public class House
{
    public int area;
    public bool garage;
    public int rooms;
    public bool garden;
    public int floors;

    public int getPrice()
    {
        return area * 3000;
    }
}
```

Pierwszy obiekt - johnHouse

- Konkretny byt.
- Ma konkretne własności.

```
House johnHouse = new House();
johnHouse.area= 200;
johnHouse.garage = true;
johnHouse.rooms = 7;
johnHouse.garden = false;
johnHouse.floors = 1;

Console.WriteLine("Price: {0}",johnHouse.getPrice());
```

Po co taka konstrukcja?

- Zmniejszenie luki reprezentacji.
- Ułatwia podział pracy i współpracę między programistami.
- Lepsze grupowanie, czytelność kodu.

Rysunek 2: Lego jako model programowania obiektowego.

Czy w C# jest możliwość skompilowania kodu bez żadnej klasy?

- Odpowiedź brzmi nie. Do kompilacji potrzebna jest "widoczna" klasa ze statyczną metodą `Main`.
- Język C# jest silnie "zorientowany obiektowo".

Domyślny szablon aplikacji konsolowej w VS

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp3
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Gdzie szukać metody `Main` w projekcie typu WPF?

- Znajduje się w tzw. automatycznie generowanym kodzie.
- Ścieżka dostępu do pliku (domyślnie):
  `...\NazwaSolucji\NazwaProjektu\obj\Debug\App.g.i.cs`

```
public static void Main() {
    WpfApp2.App app = new WpfApp2.App();
    app.InitializeComponent();
    app.Run();
}
```

Definicje

Klasa – częściowa lub całkowita definicja dla obiektów. Definicja obejmuje dopuszczalny stan obiektów oraz ich zachowania. Obiekt, który został stworzony na podstawie danej klasy nazywany jest jej instancją. Klasy mogą być typami języka programowania - przykładowo, instancja klasy Owoc będzie mieć typ Owoc. Klasy posiadają zarówno interfejs, jak i strukturę. Interfejs opisuje, jak komunikować się z jej instancjami za pośrednictwem metod, zaś struktura definiuje sposób mapowania stanu obiektu na elementarne atrybuty.

Obiekt - jest to struktura zawierająca:

- dane,
- metody, czyli funkcje służące do wykonywania na tych danych określonych zadań.

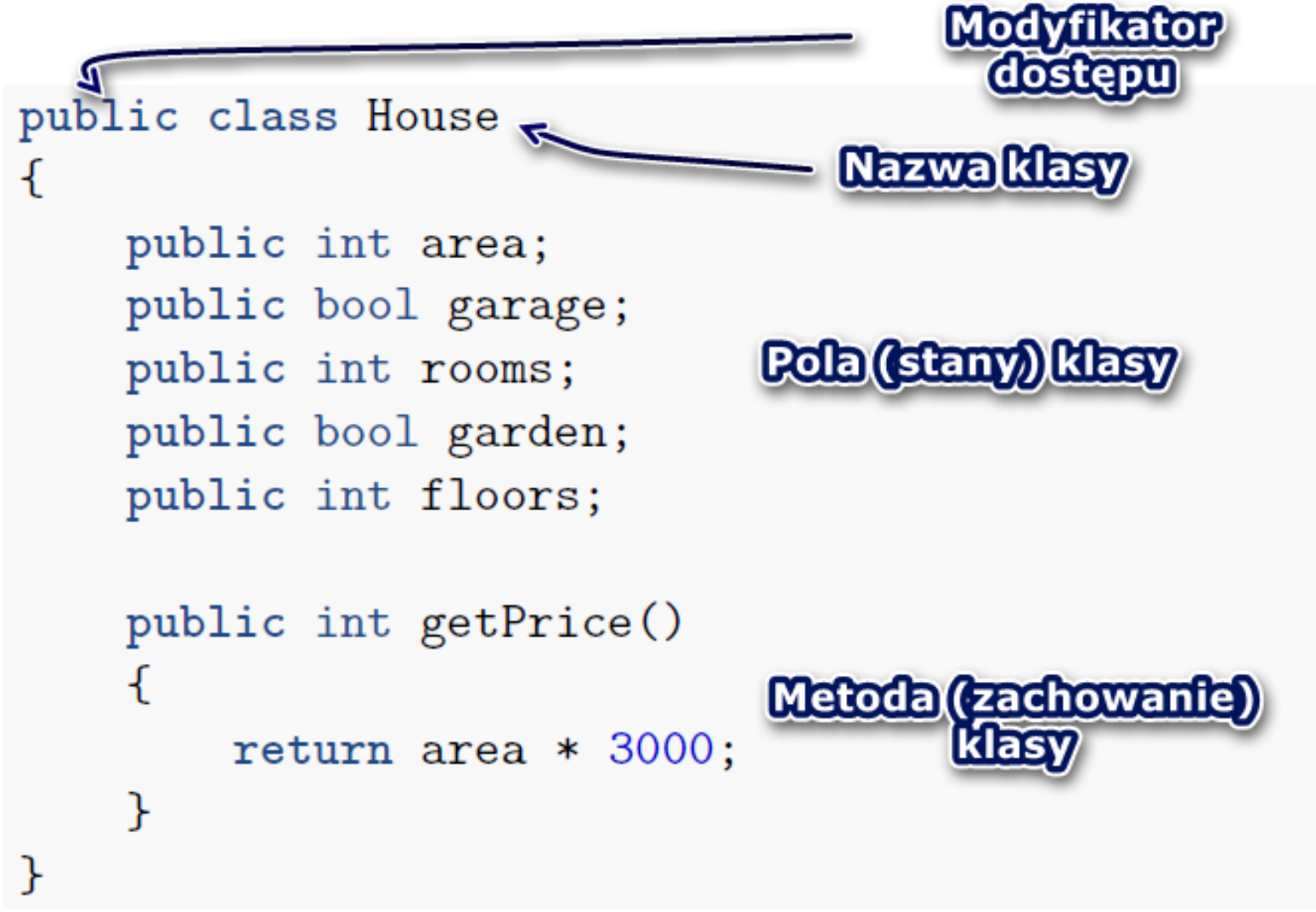
(Źródło: Wikipedia.)

Założenia/filary programowania obiektowego

- Abstrakcja - polega na ukrywaniu lub pomijaniu mało istotnych informacji a skupieniu się na wydobyciu informacji, które są niezmienne i wspólne dla pewnej grupy obiektów.
- Hermetyzacja lub inaczej mówiąc enkapsulacja polega na ukrywaniu nieistotnych informacji na temat obiektu w celu zminimalizowania efektów jego modyfikacji oraz na oddzieleniu tego co zawiera i co może zrobić obiekt od tego jak jest zbudowany i jak to robi.

Założenia/filary programowania obiektowego - c.d.

- Dziedziczenie pozwala rozszerzać możliwości klas poprzez implementacje osobnych klas rozszerzających. Dzięki dziedziczeniu możemy tworzyć nowe klasy w oparciu o już istniejące, bez potrzeby implementowania tych funkcjonalności, które zostały już zaimplementowane w klasach bazowych.
- Polimorfizm pozwala traktować różnorodne dane w ten sam sposób, w zależności od kontekstu. W trakcie wykonywania programu automatycznie są znajdowane i interpretowane odpowiednie metody w zależności od tego jak chcemy traktować nasze dane.

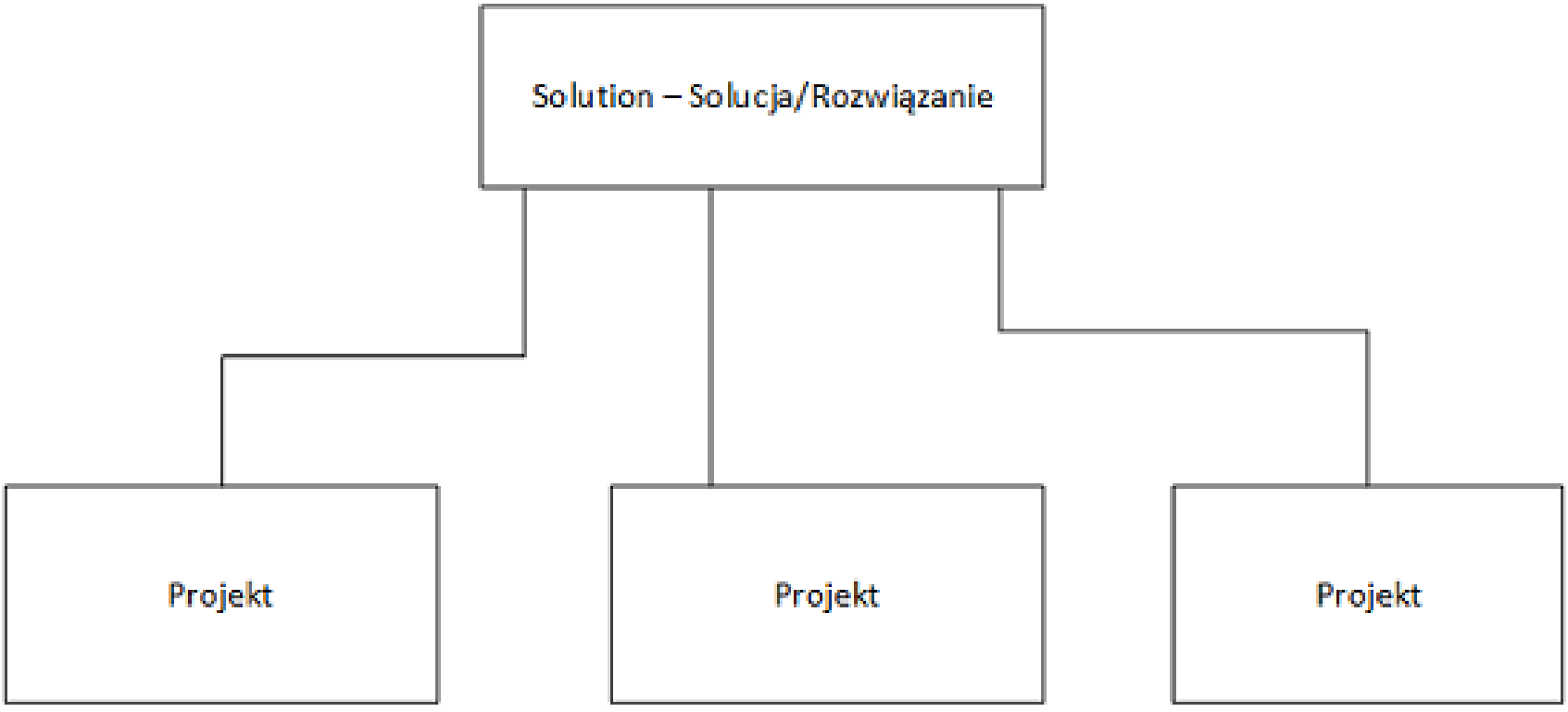


Rysunek 3: Składnia klasy w C#.

Modyfikatory dostępu

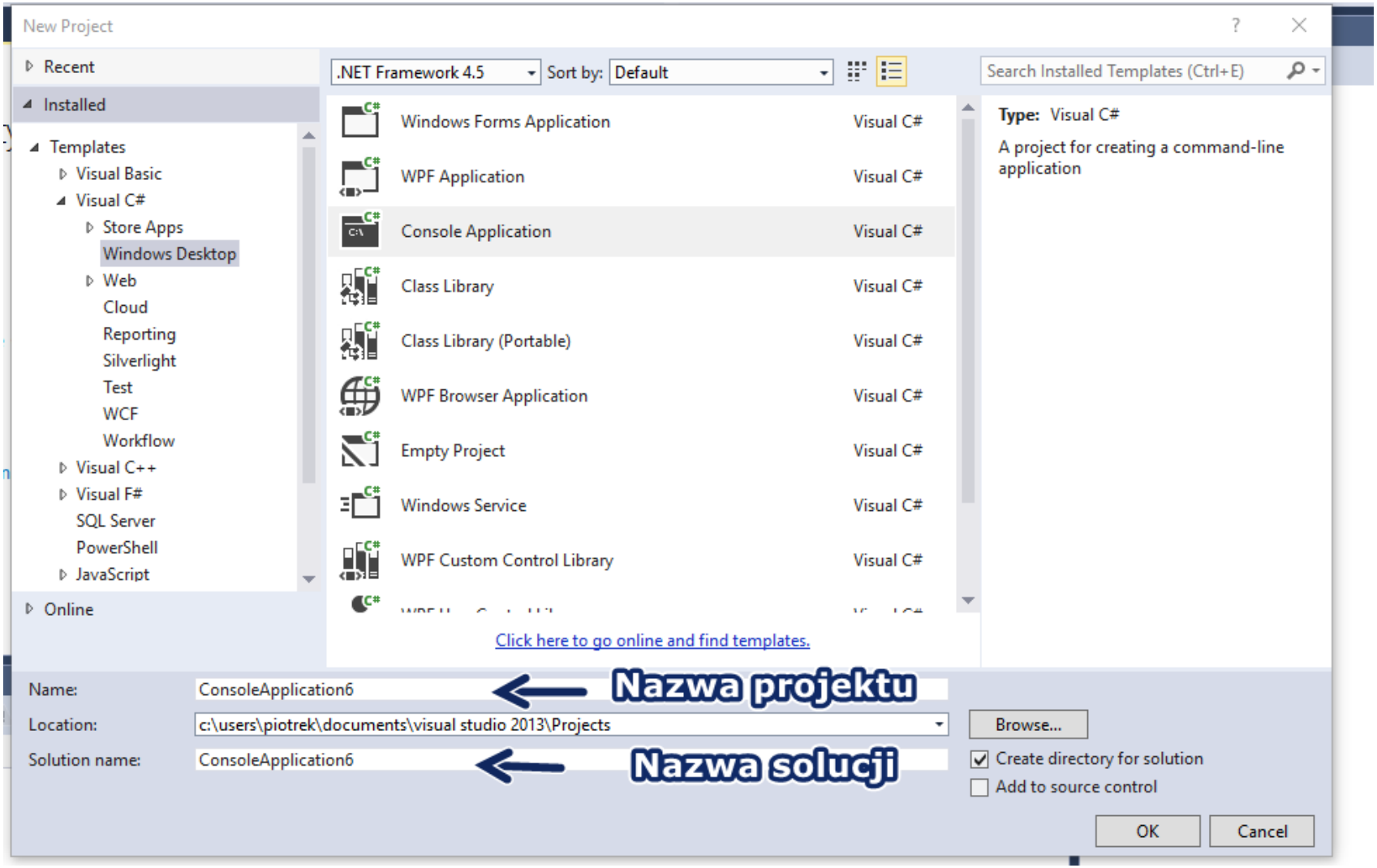
- Umożliwiają określenie dostępu do danego elementu “na zewnątrz”.
- Rodzaje:
  - publiczne (`public`),
  - chronione (`protected`),
  - wewnętrzne (`internal`),
  - wewnętrzne chronione (`protected internal`),
  - prywatne (`private`),
  - prywatne chronione (`private protected`) - od wersji 7.2 (!).

Struktura logiczna aplikacji/programu w VS

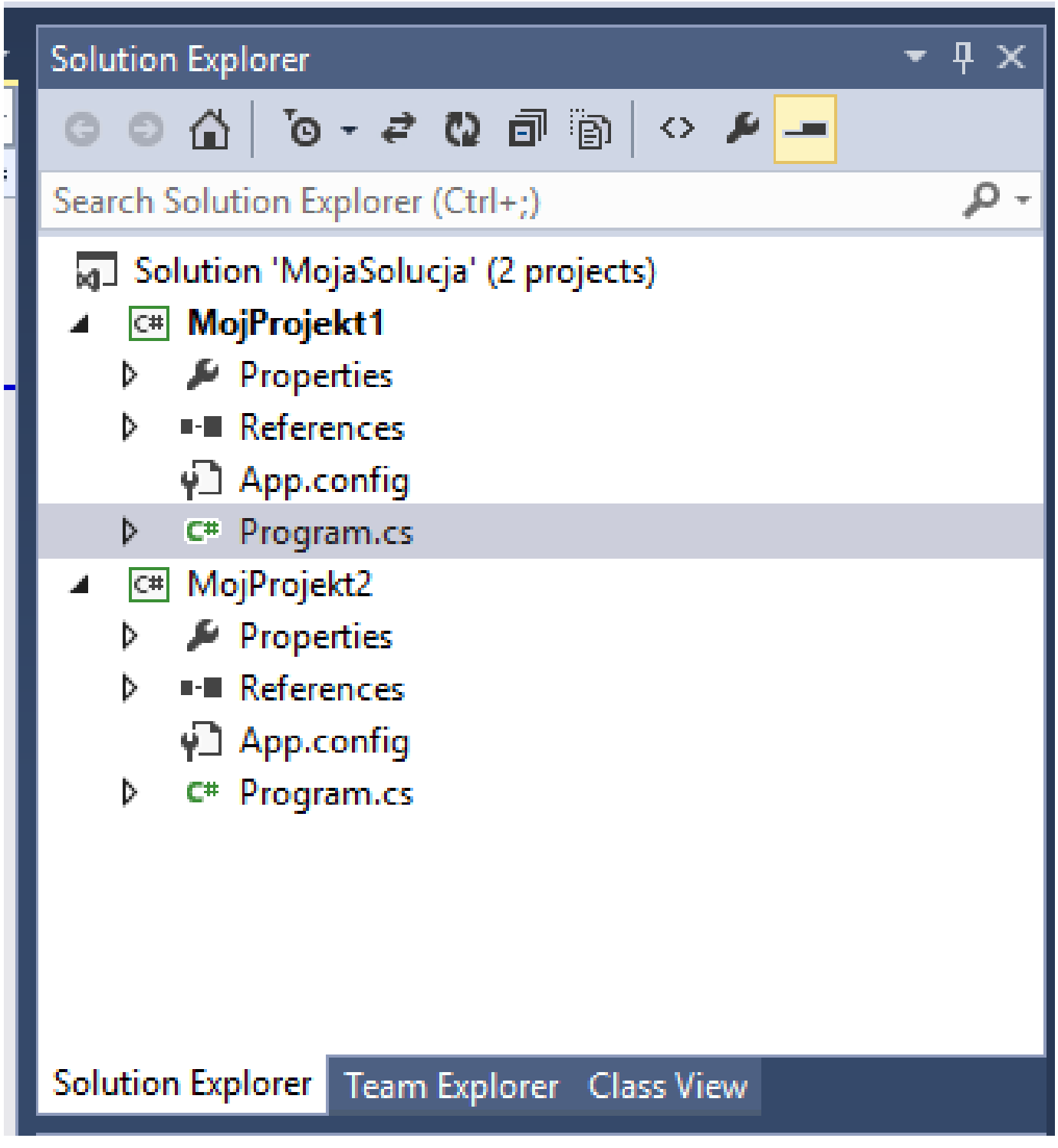


Rysunek 4: Struktura logiczna

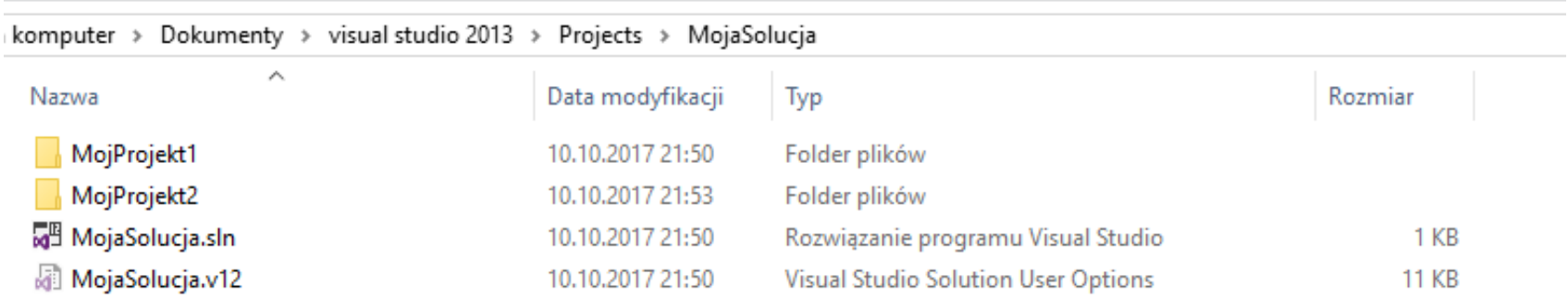
Przy tworzeniu nowego projektu, możemy wybrać oddzielną nazwę dla solucji:

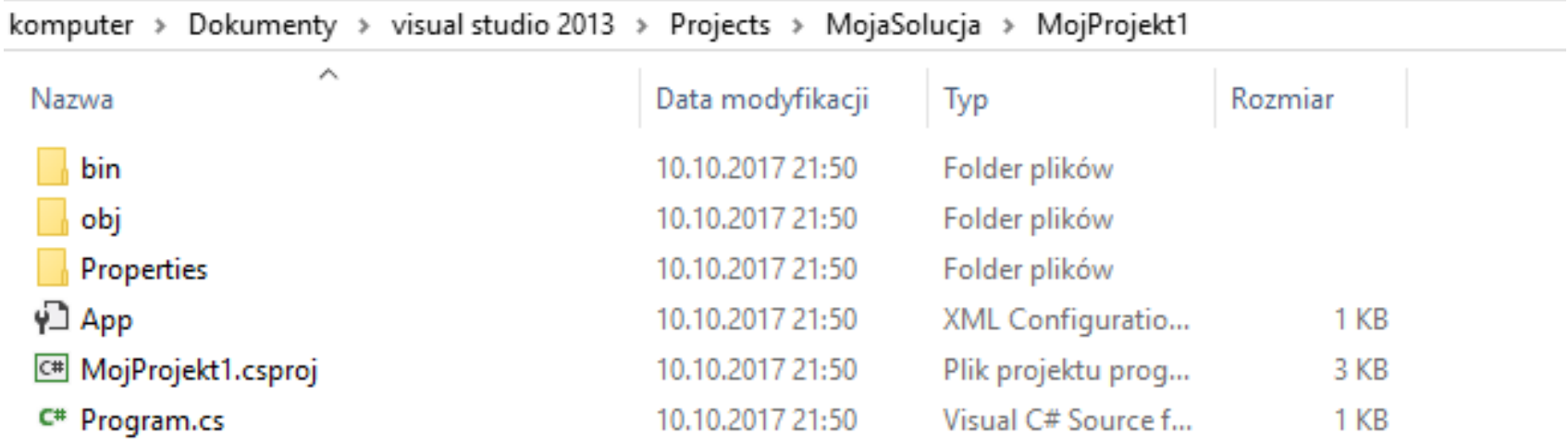


Rysunek 5: Tworzenie


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'MojaSolucja' (2 projects)
MojProjekt1
Properties
References
App.config
Program.cs
MojProjekt2
Properties
References
App.config
Program.cs
Solution Explorer
Team Explorer
Class View

Struktura fizyczna plików dla aplikacji konsolowej i domyślnego szablonu:





Modyfikatory dostępu

- `public` - elementy publiczne, dostępnymi na zewnątrz klasy,
- `private` – elementy prywatne, a dostęp do nich z poza klasy, będzie niemożliwy,
- `internal` - nie będą udostępniane na zewnątrz podzespołu (projektu),
- `protected` – elementy traktowane jako chronione, będą dostępne dla danej klasy oraz na potrzeby klas dziedziczonych, dostępne dla typów dziedziczonych na zewnątrz klasy
- `private protected` – elementy traktowane jako chronione, będą dostępne dla danej klasy oraz na potrzeby klas dziedziczonych, niedostępne dla typów dziedziczonych na zewnątrz klasy.

Przykład na rozróżnienie dwóch ostatnich będzie omówiony po dziedziczeniu (na kolejnym wykładzie).

Pola w klasie

Pola to inaczej zmienne w danej klasie.

Deklaracja:

```
modyfikator typ_danych nazwaPola;
```

W przeciwieństwie do zmiennych wewnątrz funkcji pole, jeżeli nie zostanie zainicjowane, po utworzeniu obiektu danej klasy automatycznie przyjmie swoją wartość domyślną, odpowiednio:

- dla typów liczbowych będzie to `0`,
- dla łańcucha pusty ciąg oznaczany jako "",
- dla typu boolean wartość `false`,
- dla typów referencyjnych `null`.

Metody

Metody definiują zachowanie, umożliwiając im wykonywanie konkretnych zadań, nie można ich definiować poza ciałem klasy.

Deklaracja:

- modyfikator,
- typ zwracany przez metodę,
- nazwa metody,
- lista typów i nazw parametrów metody.

Modyfikatory metod (poza modyfikatorami dostępu)

- `virtual`
- `override`
- `new`
- `sealed`
- `abstract`
- `static` - może być wywoływana bez konieczności tworzenia obiektu
- `extern` – implementacja zewn.

Dokładniejsze omówienie będzie na kolejnych wykładach.

Przykłady metod

```
public int GetPrice()
{
    return area * 3000;
}

public int GetSum(int n1, int n2)
{
    return n1 + n2;
}

public void ShowName()
{
    Console.WriteLine(name);
}
```

Czemu w C# używa się określenie metody, a nie funkcje?

- Matematyczna definicja funkcji:

Funkcja – dla danych dwóch zbiorów X i Y przyporządkowanie każdemu elementowi zbioru X dokładnie jednego elementu zbioru Y.

- Metody mogą być "szerzej rozumiane". Mamy metody bez parametru (czyli bez zbioru X) i metody bez typu zwracanego (`void`, czyli bez zbioru Y).

Przeciążanie metod

Przeciążone metody to takie, które:

- mają taki sam typ zwracany,
- mają identyczną nazwę,
- różnią się tylko ilością lub typami parametrów.

```
static string GetType()
{
    return "Metoda bez argumentów";
}
static string GetType(int x)
{
    return "Liczba całkowita";
}
```

Operator `new`/Konstruktor

Operator `new` automatycznie wywołuje konstruktor.

Konstruktor:

- metoda wykonywana zaraz przy tworzeniu instancji obiektu,
- zawiera modyfikatory dostępu i parametry,
- nazwa pokrywa się z nazwą klasy.

Konstruktor domyślny

```
public House()
{

}
```

Konstruktor domyślny z wartościami:

```
public House()
{
    area = 150;
}
```

Konstruktor z parametrami

```
public House(int area)
{
        this.area = area;
}
```

Deklaracja obiektu z parametrem

```
House johnHouse = new House(250);
House tomHouse = new House() { area ="Ford" };
```

Klasa nie musi posiadać konstruktora. Modyfikator, słowo kluczowe lub parametry są jedynie opcją. Jednak określenie parametru wpływa na sposób tworzenia obiektu. Np. jeśli mamy w klasie konstruktor tylko z jednym parametrem, to nie możemy tworzyć obiektu konstruktorem domyślnym (bez parametru). Dlatego często korzystamy z tzw. przeciążenia konstruktorów.

```
class Test
{
      public Test()
      {
      }

      public Test(string param1)
      {
      }
}
```

Uwaga! Dobrą praktyką jest umieszczanie minimalnej instrukcji w konstruktorze. Pod kątem kompilacji poprawnie możemy tu dodać również inne metody, jednak powinniśmy rozdzielić zadania dla poszczególnych metod.

Po co przydają się konstruktory?

- Nie trzeba wywoływać dodatkowych metodą, inicjują stan klasy na początku.
- Lepsza czytelność i zwięzłość kodu.

Prywatny konstruktor?

- Dopuszczalnym (pod kątem kompilacji) jest stworzenie prywatnego konstruktora. Aby wtedy z niego skorzystać "na zewnątrz" klasy potrzeba jest statyczna metoda, w ramach której będzie wywołany prywatny konstruktor. Dokładniej zostanie to omówione przy wzorcach projektowych.

```
class Test
    {
        private Test()
        {

        }

        public static Test Create()
        {
            return new Test();
        }
    }
```

```
Test sw = Test.Create();
```

Klasa na diagramie UML

UML - UML (Unified Modelling Language) - graficzny system wizualizacji, specyfikowania oraz dokumentowania składników systemów informatycznych.

Nazwa klasy:

```
modyfikator NazwaKlasy ustawienia
```

Pola klasy:

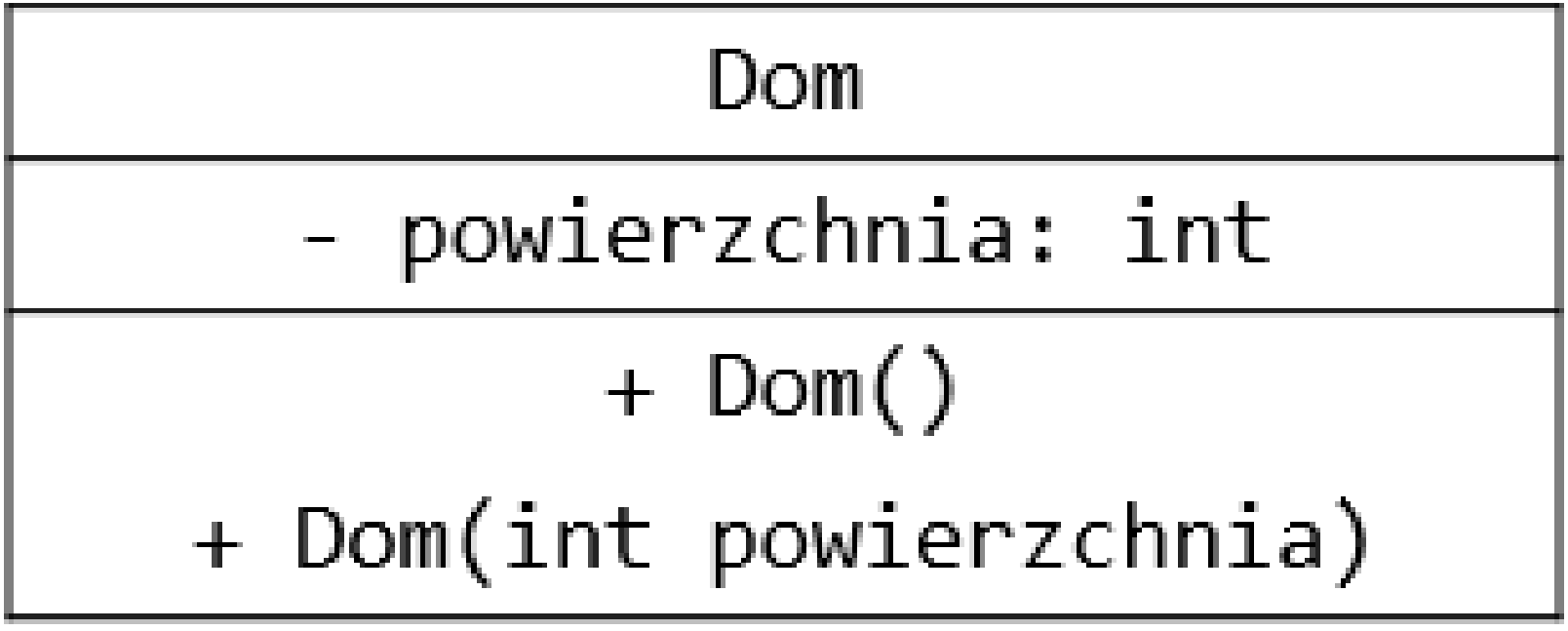
```
modyfikator nazwaPola: typ =   wartoscPoczatkowa
ustawienia
```

Metody

```
modyfikator NazwaMetody(parametr1,     ...):     typ
ustawienia
```

Modyfikatory dostępu:

- `+` to `public`
- `#` to `protected`
- `-` to `private`
- `~` to `internal`



Rysunek 7: Diagram UML klasy.

Przykład:

```
class Dom
{
    private int powierzchnia;
    public Dom()
    {
        powierzchnia = 150;
    }

    public Dom(int powierzchnia)
    {
        this.powierzchnia = powierzchnia;
    }
}
```

Hermetyzacja

Hermetyzacja:

- Zapewnia utrzymanie właściwego stanu klasy. Pola klasy są inicjowane, modyfikowane i odczytywane w sposób przewidziany przez autora klasy.
- Ukrycie wewnętrznego stanu klasy, a udostępnienie tylko tego co jest niezbędne do komunikacji z innymi klasami
- Użycie odpowiednich modyfikatorów dostępu do pól i metod,
- Innymi słowy, nie pozwalajcie innym mieszać w stanie waszej klasy, gdyż to utrudnia odnajdywanie błędów.

Przykładowe zadanie polega na napisaniu klasy `Licznik`, ma być ona wykorzystywana w wielu modułach zliczających rozpoczęcie i zakończenie jakiegoś zdarzenia. Licznik nie może być mniejszy od 0. Klasa `Licznik` ma 2 metody `Zwiększ()`, `Zmniejsz()`.

```
class Licznik
{
    public int stan;

    public void Zwieksz()
    {
        this.stan += 1;
    }

    public void Zmniejsz()
    {
        if (this.stan > 0)
            this.stan -= 1;
    }
}
```

```
Licznik l1 = new Licznik();
l1.Zmniejsz();
l1.Zmniejsz();
l1.stan = -200;
l1.Zwieksz();
```

Co należałoby poprawić w klasie `Licznik`?

Najlepszym wyjściem byłoby uniemożliwienie zmiany pola `stan`, ustalając jego dostęp z `public` na `private`. Dzięki temu nikt nie może dostać się do zmiennej i jest to zablokowane na poziomie kompilacji.

{Źródło: https://drive.google.com/drive/folders/0B4rqQMWTVxb8WTVkS2kySTRpN1k?usp=sharing .}

Słowo kluczowe `static`

Metoda statyczna może być używana bez tworzenia instancji klasy. Podstawowym przykładem jest metoda `Main` w klasie `Program`.

```
class Program
{
    static void Main(string[] args)
    {
    }
}
```

Innymi statycznymi metodami są metody z klasy `Console` jak `ReadLine`, `ReadKey` i `WrtieLine`...

Pola statyczne mają wspólną wartość dla wszystkich instancji danej klasy. Przydają się w dosyć specyficznych sytuacjach jak np. posiadanie pewnych "zmiennych globalnych" czy zliczanie instancji danego typu.

```
public static double Pi = 3.14;
```

Jeśli chcemy "zabezpieczyć" klasę przed tworzeniem jej instancji, możemy oznaczyć ją jako statyczną. Przykładem jest klasa `Console`.

Ważne!

- Elementy statyczne mogą odnosić się do innych elementów statycznych.
- Wewnątrz klasy statycznej pola i metody muszą być statyczne.
- Wewnątrz metody statycznej metody możemy używać tylko pól statycznych.

Konstruktor statyczny

- nie może przyjmować żadnych argumentów ani nie może być przy nim żadnych modyfikatorów
- wywoływany jest przed wszystkimi innymi konstruktorami
- w danej klasie może znajdować się maksymalnie jeden konstruktor
- jest wywoływany podczas tworzenia instancji (dla klas niestatycznych) lub podczas pierwszego dostępu do statycznej składowej.

```
class Matematyka
{
    public static double Pi;

    static Matematyka()
    {
        Matematyka.Pi = 3.14;
    }
}
```

Słowo kluczowe `const`

Pola stałe muszą być zainicjowane i nie mogą być zmieniane w trakcie wykonywania programu.

```
const double Pi = 3.14;
```

Destruktor

- Język C# automatycznie odzyskuje pamięć (nie trzeba jawnie usuwać obiektów).
- Czasem jednak chcemy samodzielnie zarządzać zasobami pamięci – w tym celu mamy destruktor.

```
~NazwaKlasy
{
    instrukcja do wykonania przy niszczeniu;
}
```

Właściwości

Właściwości (ang. property) - niektórzy po polsku nazywają je propercjami.

Do tej pory składnia naszego kodu wyglądała następująco:

```
class Osoba
{
    private int wiek;
    public void UstawWiek(int wiek) { this.wiek = wiek; }
    public int PobierzWiek() { return wiek; }
}
```

W metodzie `Main` klasy `Program` wyglądało to tak:

```
Osoba osoba1 = new Osoba();
osoba1.UstawWiek(21); //długo
```

A nie było dostępu do pól bezpośrednio z uwagi na hermetyzację:

```
osoba1.wiek = 21; //niedostępne
```

Czym są zatem właściwości?

- specjalna konstrukcja języka C# pozwalająca połączyć zachowanie pól i metod
- wyglądają jak pola więc możemy pobrać ich wartość bądź je przypisać, a jednocześnie zachowują się jak metody czyli możemy stosować je w interfejsach

Składnia właściwości:

```
private int wiek; //pole
public int Wiek //właściwość
{
    get { return wiek; }
    set { wiek = value; }
}
```

Skrócona deklaracja:

```
class Osoba
{
    public int Wiek { get; set; }
}
```

Właściwość tylko do odczytu:

```
private int wiek;
public int Wiek
{
    get
    {
        return wiek;
    }
}
```

Właściwość tylko do zapisu:

```
private int wiek;
public int Wiek
{
    set
    {
        wiek = value;
    }
}
```

Dobra praktyka wg MSDN:

- ilość kodu/instrukcji umieszczonych we właściwościach powinniśmy ograniczyć do minimum
- dopuszczalne są jedynie „proste” operacje

Dostępność właściwości:

- domyślne powielany jest modyfikator dostępu właściwości dla get i set, ale można go nadpisać

```
public int Wiek
{
    private get { return wiek; }
    set { wiek = value; }
}
```

- nie możemy nadpisać obu jednocześnie (lepiej zmienić modyfikator właściwości)

```
public int Wiek
{
   private get { return wiek; }
   private set { wiek = value; }
}
```

Powyższy kod skutkuje błędem kompilacji.

- modyfikatory dla `get` i `set` nie mogą być bardziej dostępne niż modyfikator samej właściwości

```
private int Wiek
{
    public get { return wiek; }
    set { wiek = value; }
}
```

Pola czy właściwości – co używać?

- właściwości nie mogą być użyte przy słowach kluczowych `ref` i `out`

```
Osoba osoba1 = new Osoba();
int.TryParse("21", out osoba1.Wiek);
```

- właściwości nie mogę być stałymi
- możemy także zainicjować obiekt dzięki właściwościom

```
Osoba osoba1 = new Osoba() { Wiek = 22 };
```

Operator przypisania = w C#?

```
int a = 6;
int b = a;
Console.WriteLine("a={0}, b={1}", a, b);
b++;
Console.WriteLine("Po zmianie");
Console.WriteLine("a={0}, b={1}", a, b);
```

Na konsoli otrzymamy:

```
a=6, b=6
Po zmianie
a=6, b=7
```

```
class Osoba
{
    private int wiek;
    public Osoba() { }
    public Osoba(int wiek)
    {
        this.wiek = wiek;
    }
    public void UstawWiek(int wiek)
    {
        this.wiek = wiek;
    }
    public int PobierzWiek()
    {
        return wiek;
    }
}
```

```
Osoba o1 = new Osoba(30);
Osoba o2 = new Osoba();
o2 = o1;
Console.WriteLine("Wiek 1os. {0}", o1.PobierzWiek());
Console.WriteLine("Wiek 2os. {0}",o2.PobierzWiek());
o1.UstawWiek(31);
Console.WriteLine("Po zmianie");
Console.WriteLine("Wiek 1os. {0}", o1.PobierzWiek());
Console.WriteLine("Wiek 2os. {0}", o2.PobierzWiek());
if (Object.ReferenceEquals(o1, o2))
    Console.WriteLine("Ten sam obiekt");
else
    Console.WriteLine("Różne obiekty");
```

Wynik na konsoli:

```
Wiek 1os. 30
Wiek 2os. 30
Po zmianie
Wiek 1os. 31
Wiek 2os. 31
Ten sam obiekt
```

Dlaczego tak się dzieje?

W C#:

- typy wartościowe przypisywane są przez wartość
- typy referencyjne przypisywane są przez referencję

Uwaga:

- używanie odpowiednich sformułować zależy od języka. Np. w Pythonie przypisywanie też jest przez referencję (która jest nieco inaczej rozumiana), a “wszystko jest obiektem”.

Python:

```
a=5
b=a
b+=2
print(a)
print(b)
```

```
## 5
## 7
```

Dziedziczenie

Dziedziczenie

- „Mechanizm”, dzięki któremu jedna z klas może posiąść cechy innej klasy.
- „cechy” – `public`, `protected`, `internal`, `protected internal`, ~~`private`~~
- Klasa bazowa – klasa, z której jest dziedziczone
- Klasa potomna/pochodna – klasa, która dziedziczy

Rozważmy przykład:

- Rozważmy mamy trzy klasy `Pojazd`, `Rower`, `Samochod`.
- Samochód jest Pojazdem.
- Rower jest Pojazdem.

Po co jest dziedziczenie?

- Klasy potomne mogą współdzielić zachowania klas potomnych.
- Możemy rozszerzyć klasy bez powielania kodu.
- Uwypukla wspólne cechy (wspiera abstrakcję).

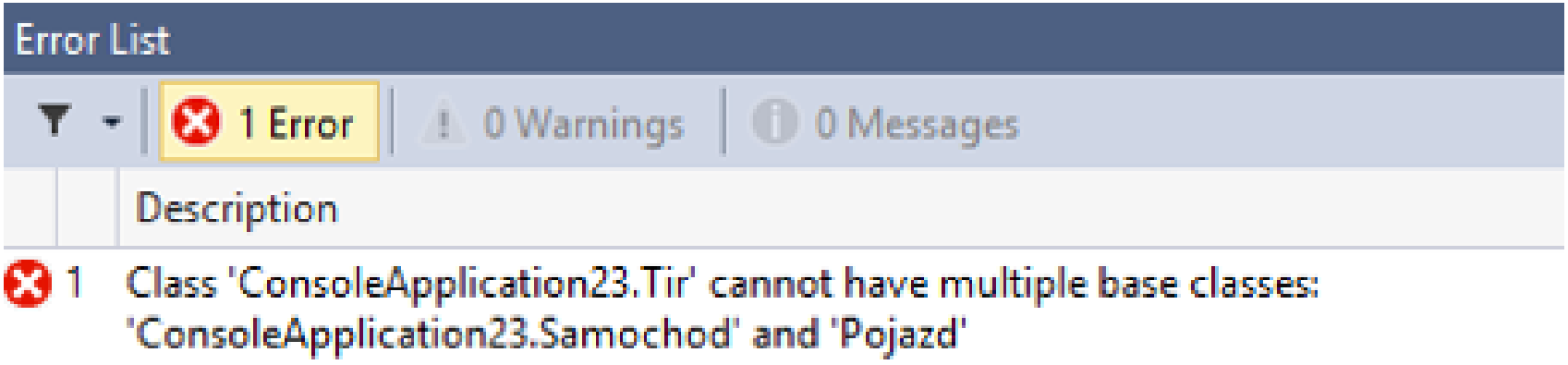
```
class Bazowa
{
    public int pole;
    public void Metoda1(){ }
}

class Pochodna : Bazowa
{
    public int Metoda2()
    {
        return pole * 2;
    }
}
```

Wielodziedziczenie klas?

Wielodziedziczenie klas – dziedziczenie z kilku klas bazowych jednocześnie. W języku C# jest to nie możliwe.

Inne przykładowe języki programowanie umożliwiające wielodziedziczenie: C++, Perl, Python.

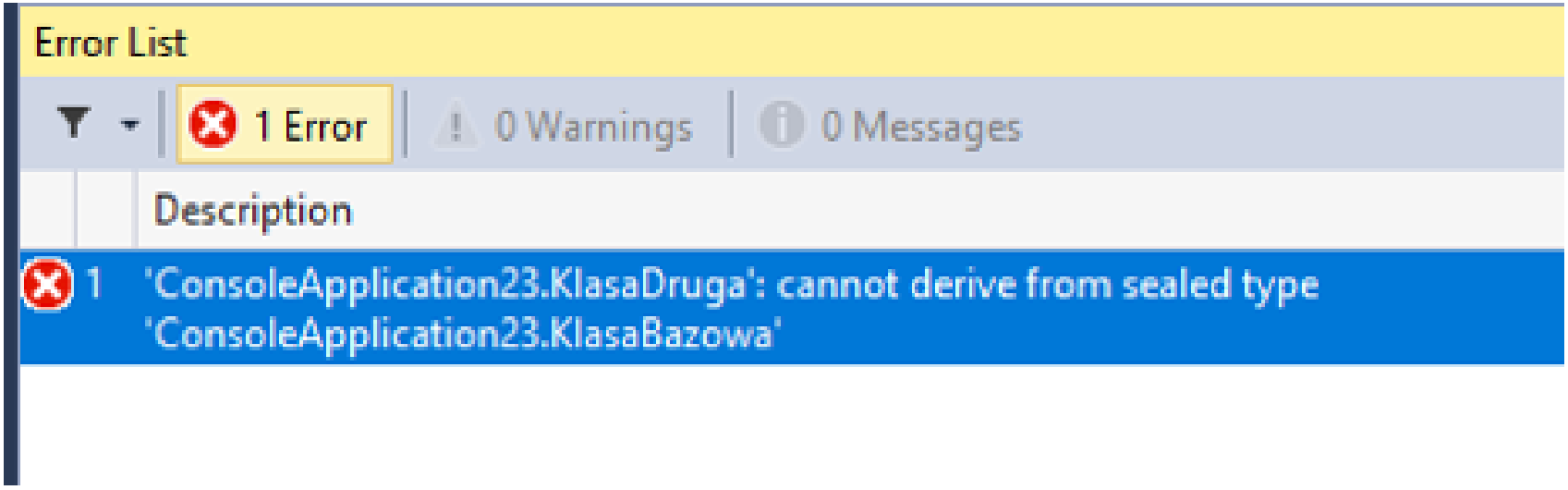
```
class Pojazd
{
    // elementy klasy
}
class Samochod : Pojazd
{
    // elementy klasy
}
class Tir : Samochod, Pojazd
{
    // elementy klasy
}
```


Error List
1 Error
0 Warnings
0 Messages
Description
1
Class 'ConsoleApplication23.Tir' cannot have multiple base classes:
'ConsoleApplication23.Samochod' and 'Pojazd'

Klasy zaplombowane, finalne

- Klasy z modyfikatorem dostępu sealed są traktowane jako „zaplombowane”.
- Nie mogą być klasami bazowymi dla innych - nie można z nich dziedziczyć.

```
public sealed class KlasaBazowa
{
    // elementy klasy
}

public class KlasaDruga : KlasaBazowa
{
    // elementy klasy
}
```


Error List
1 Error
0 Warnings
0 Messages
Description
1 'ConsoleApplication23.KlasaDruga': cannot derive from sealed type 'ConsoleApplication23.KlasaBazowa'

- Każda klasa w C# dziedziczy niejawnie z klasy `Object`.
- Wszystkie klasy w .NET dziedziczą po niej.

Konstruktory a dziedziczenie

- Konstruktory nie podlegają dziedziczeniu.
- W każdej klasie konstruktor należy napisać na nowo.
- Za pomocą inicjatora base możemy wywołać konstruktor klasy bazowej.

```
class Pojazd
{
    protected string marka;
    public Pojazd(string marka)
    {
        this.marka = marka;
    }
}
class Samochod : Pojazd
{
    int iloscKol;
    public Samochod(string marka, int iloscKol)
        : base(marka)
    {
        this.iloscKol = iloscKol;
    }
}
```

Rzutowanie a dziedziczenie

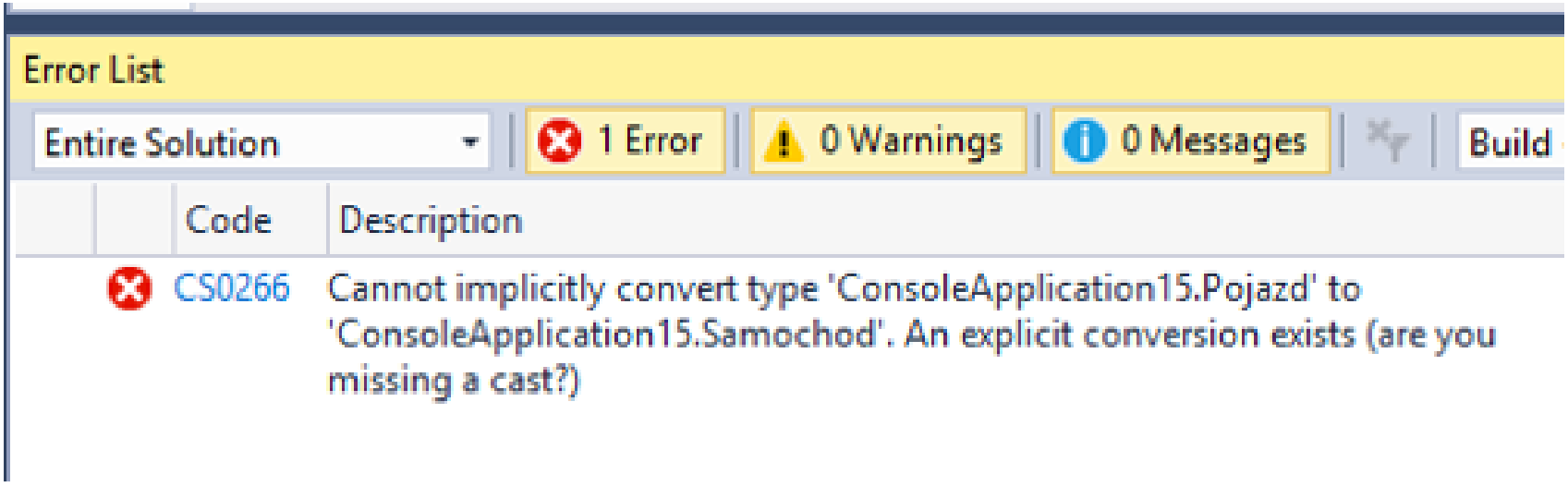
- Rzutowanie w górę (zawsze możliwe i skuteczne).
  - Samochód możemy traktować jak Pojazd.
- Rzutowania w dół (możliwe gdy rzeczywiście obiekt jest typu pochodnego).
  - Nie każdy Pojazd jest Samochodem.
  - Nie każdy prostokąt jest kwadratem.

Rzutowanie w górę

```
//obiekt klasy potomnej
Samochod a1 = new Samochod();
//rzutowanie w górę
Pojazd a2 = a1;
```

Rzutowanie w dół

```
Samochod a1 = new Samochod();
Pojazd a2 = a1;
Samochod a3 = a2
```



Jak to naprawić? I sposób - jawny rzut

```
Samochod a1 = new Samochod();
Pojazd a2 = a1;
Samochod a3 = (Samochod)a2;
```

Ale mamy ryzyko błędu:

```
Pojazd p1 = new Pojazd();
Samochod p2 = (Samochod)p1;
```

II sposób - bezpieczne rzutowanie - `is` - zwraca `true` jeśli lewa strona obiektu może zostać rzutowana na typ określony po prawej stronie.

```
Pojazd p1 = new Samochod();
if (p1 is Samochod)
{
    Samochod p2 = (Samochod)p1;
}
```

III sposób: C# 7.0 - pattern matching

```
Pojazd p1 = new Samochod();
if (p1 is Samochod p2)
{
        Console.WriteLine(Object.ReferenceEquals(p1, p2));
}
```

IV sposób: `as` - wykona rzutowanie, jeśli jest możliwe. Jeśli nie, zwróci wartość `null`.

```
Pojazd p1 = new Pojazd();
Samochod p2 = p1 as Samochod;
```

Modyfikatory dostępu w dziedziczeniu

- `protected` - elementy dostępne dla klas pochodnych
- `protected internal` - elementy dostępne dla klas pochodnych lub w ramach projektu
- `private protected` - dostępne dla klas pochodnych w tym samym projekcie

Interfejsy

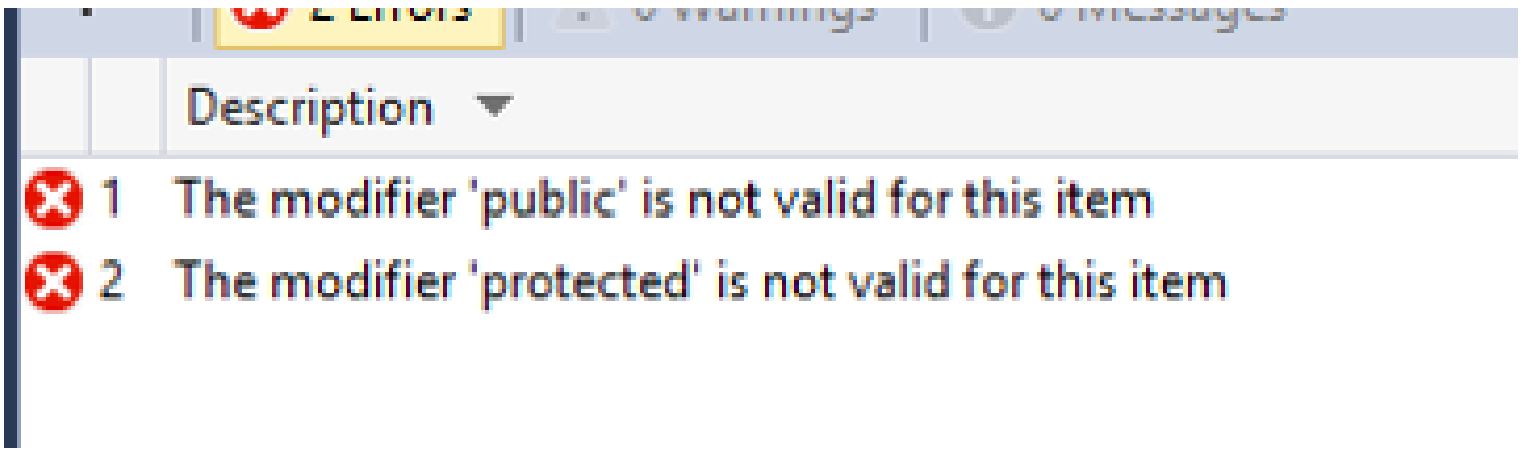
Definicja interfejsu wg wikipedii

Interfejs – definicja abstrakcyjnego typu posiadającego jedynie operacje, a nie dane. Kiedy w konkretnej klasie zdefiniowane są wszystkie metody interfejsu mówimy, że klasa implementuje dany interfejs. W programie mogą być tworzone zmienne typu referencja do interfejsu, nie można natomiast tworzyć obiektów tego typu. Referencja może wskazywać na obiekt dowolnej klasy implementującej dany interfejs. Interfejs określa udostępniane operacje, nie zawiera natomiast ich implementacji i danych. Z tego powodu klasy mogą implementować wiele interfejsów, bez problemów wynikających z wielokrotnego dziedziczenia. Wszystkie metody w interfejsie z reguły muszą być publiczne.

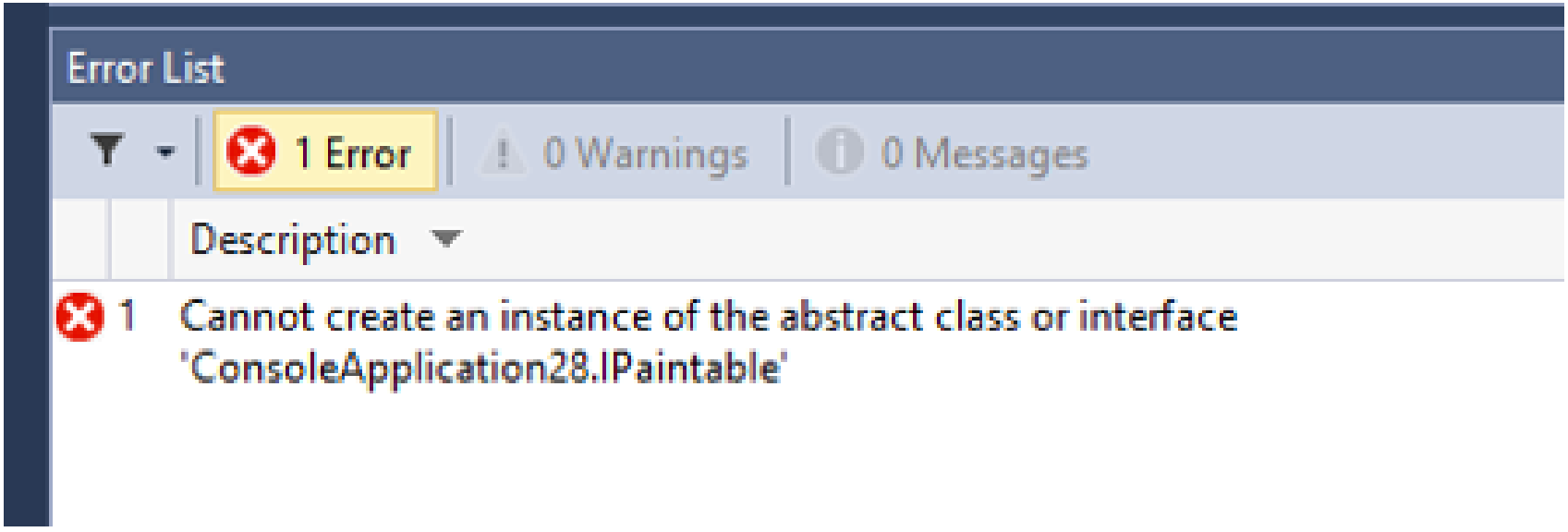
```
interface IPaintable
{
    void Maluj();
}

interface IPaintable
{
    void Maluj(string kolor);
}
```

Dodanie modyfikatora skutkuje błędem kompilacji.

```
interface IPaintable
{
    public void Maluj();
    protected void Maluj(string
}
```



```
IPaintable sth = new IPaintable();
```



- Klasa musi implementować wszystkie metody z interfejsu.
- Klasa może dziedziczyć po kilku interfejsach (ale tylko po jednej klasie).
- Jeśli klasa ma podpięty interfejs pochodny z innego interfejsu bazowego, to klasa musi implementować wszystkie metody z obu interfejsów.
- Możemy rzutować obiekt na interfejs - ale wtedy dostępne będą dla niego tylko metody z interfejsu.

Jawna i niejawna implemetacja interfejsu w C#

W języku C# możemy wyróżnić tzw. jawną (explicite) i niejawną (implicit) implementacją interfejsu.

```
interface ISport
{
    void Graj();
}
class Osoba : ISport
{
    public void Graj() {} //niejawna
    void ISport.Graj() //jawna
    {
        //jakiś kod
    }
}
```

Przykłady implementacji systemowych interfejsów

- `IComparable` – gist.
- `IComparable<T>` - gist.

Temat klonowania, kopiowania a interfejs `ICloneable` - gist.

Polimofizm

Definicja polimorfizmu wg wikipedii

Polimorfizm (z gr. wielopostaciowość) – mechanizmy pozwalające programiście używać wartości, zmiennych i podprogramów na kilka różnych sposobów. Inaczej mówiąc jest to możliwość wyabstrahowania wyrażeń od konkretnych typów.

Polimorfizm w C#

- statyczny
  - przeciążenie funkcji
  - przeciążenie operatorów
- dynamiczny
  - funkcje wirtualne
  - funkcje abstrakcyjne

Napisanie metody - to nie jest polimorfizm

```
class Pojazd
{
    public void Jedz()
    {
        Console.WriteLine("Jadę pojazdem");
    }
}
class Samochod: Pojazd
{
    public void Jedz()
    {
        Console.WriteLine("Jadę samochodem");
    }
}
```

```
Pojazd p1 = new Pojazd();
p1.Jedz();
Samochod s1 = new Samochod();
s1.Jedz();
Pojazd sp1 = new Samochod();
sp1.Jedz();
```

Wykonanie kodu kończy się ostrzeżeniem kompilatora.

Podpowiedź sugeruje użycie `new` (czyli nadpisanie metody), jednak takie rozwiązanie nie jest najlepsze - nie daje elastyczności kodu.

Metoda wirtualna

- oznaczona słowem kluczowym `virtual`
- włącza mechanizm polimorfizmu dynamicznego
- tworzymy ją w klasie bazowej
- w klasach pochodnych przesłaniamy metody wirtualne za pomocą słowa kluczowego `override`
- przesłonięcie nie jest obowiązkowe, w razie jego braku zostanie wywołana metoda klasy bazowej
- metody statyczne ani prywatne nie mogą być wirtualne
- metody, które nie są wirtualne, nie można przesłonić za pomocą `override`

Po co taka konstrukcja?

- dajemy informację dla innej osoby zajmującej się kodem
- przy tworzeniu klasy potomnych mamy elastyczność: możemy zostawić to jak jest w klasie bazowej lub przesłonić

Metody wirtualne w klasie `Object`

- w C# każda klasa dziedziczy niejawnie z klasy `Object`

```
public class Object
{
    /* składowe wirtualne */
    virtual public bool Equals(object o);
    virtual protected void Finalize();
    virtual public string ToString();
    /* itp.. */
}
```

Metody abstrakcyjne

- poprzedzone słowem kluczowym `abstract`
- zdefiniowane w klasie bazowej
- nie zawierają ciała metody (podobnie jak przy interfejsach)
- mogą być zadeklarowane tylko w klasie abstrakcyjnej (poprzedzonej słowem `abstract`)
- nie możemy stworzyć egzemplarza (obiektu) klasy abstrakcyjnej (podobnie jak przy interfejsach), ale możemy dziedziczyć po klasie abstrakcyjnej
- klasa abstrakcyjna może posiadać zwykłe metody (z implementacją)
- klasa pochodna do klasy abstrakcyjnej musi przesłonić wszystkie metody abstrakcyjne