Zaawansowane programowanie obiektowe - wykład 2

dr Piotr Jastrzębski

Klasa, obiekt

Założenia/filary programowania obiektowego

- Abstrakcja
- Hermetyzacja
- Dziedziczenie
- Polimorfizm

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).

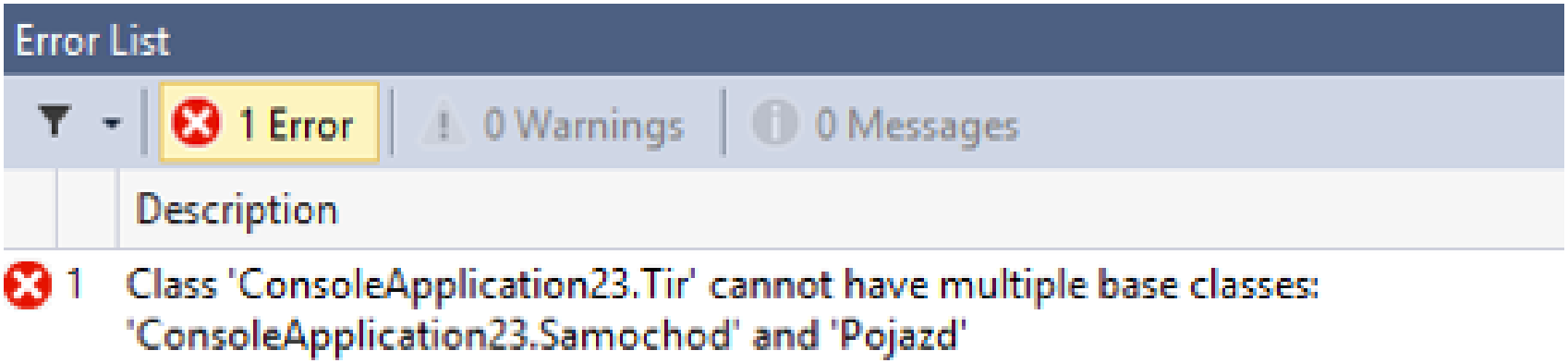
```
class Bazowa
{
    public int pole;
    public void Metoda1(){ }
}

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
}
```

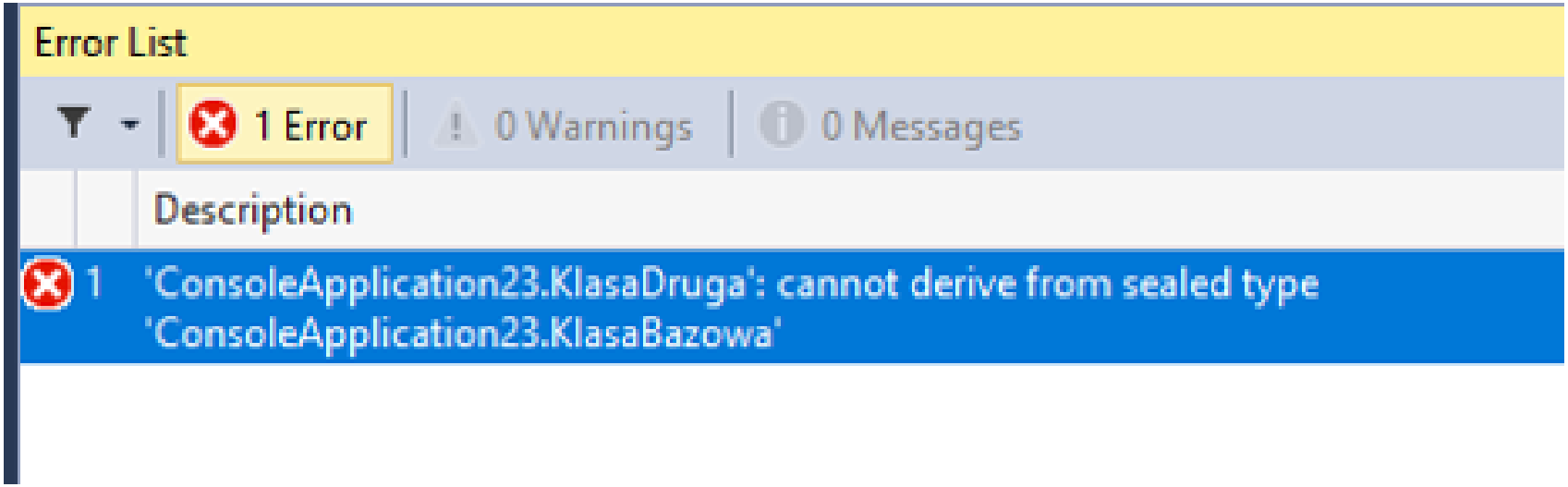

Error List
1 Error
0 Warnings
0 Messages
Description
1 Class 'ConsoleApplication23.Tir' cannot have multiple base classes: 'ConsoleApplication23.Samochod' and 'Pojazd'

Rysunek 1:

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'

Rysunek 2:

- 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

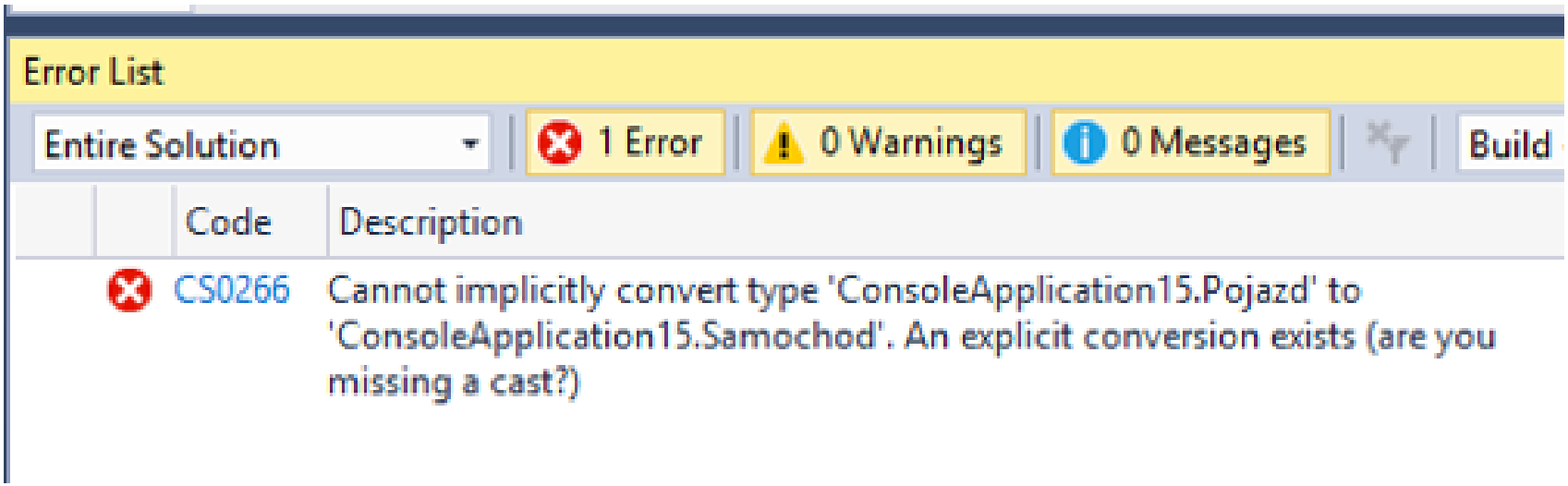
- Rzutowanie w górę (zawsze możliwe i skuteczne).
  - Samochód możemy traktować jak Pojazd.
- Rzutowania w dół (możliwe gdy rzeczywiście obiekt jest typu pochodnego).
  - Nie każdy Pojazd jest Samochodem.
  - Nie każdy prostokąt jest kwadratem.

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
```



Rysunek 3:

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

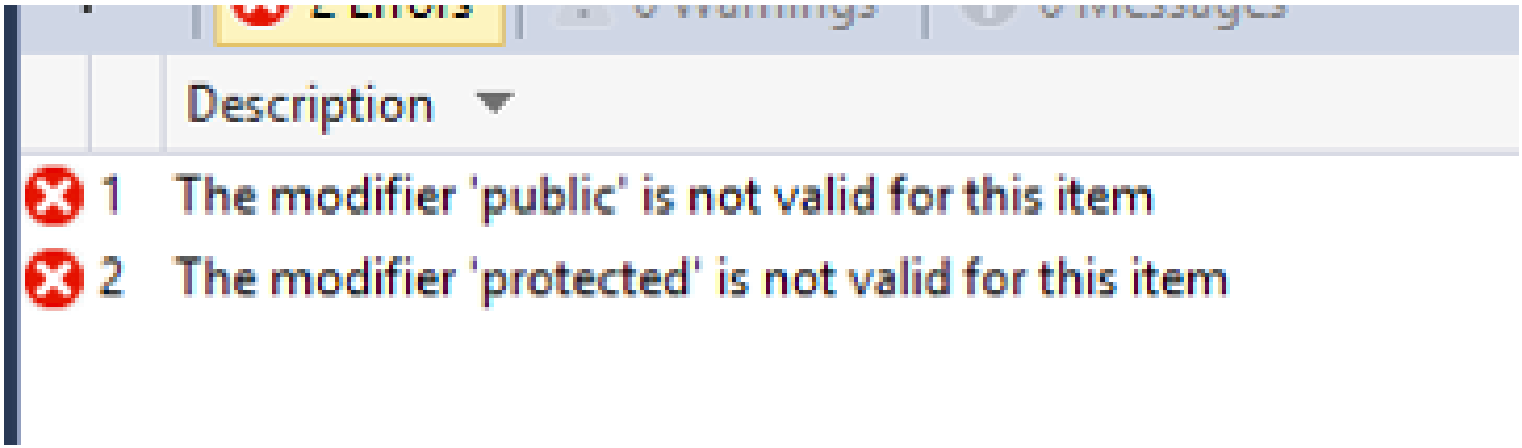
Definicja interfejsu wg wikipedii

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);
}
```

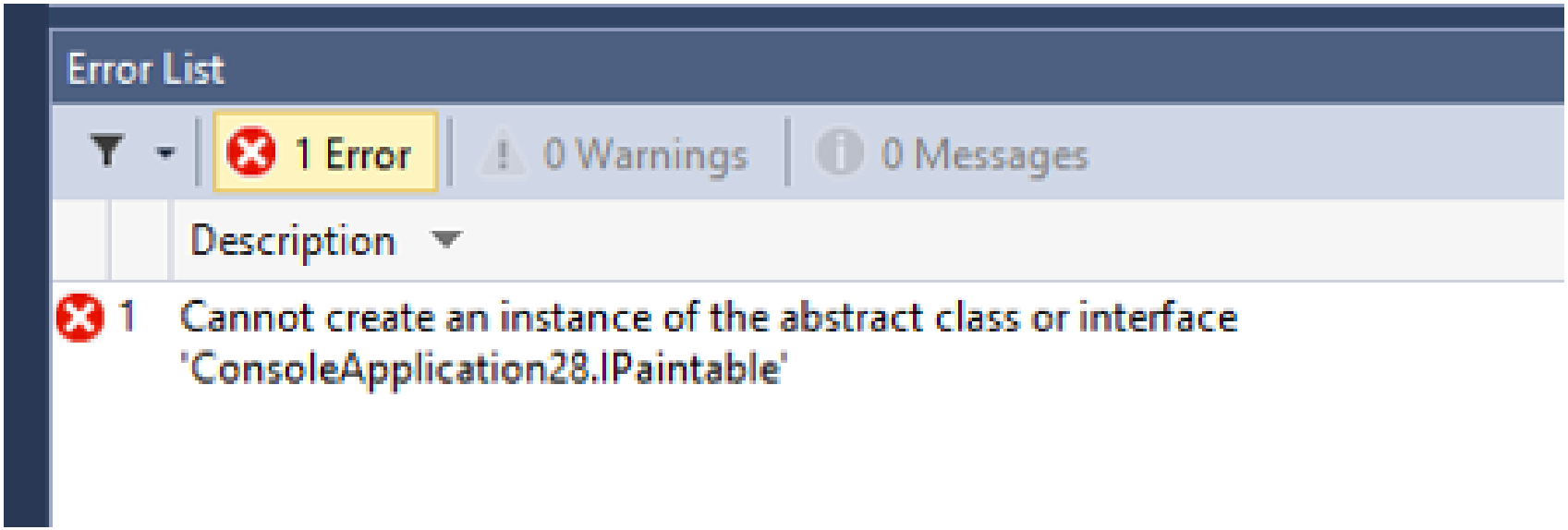
Dodanie modyfikatora skutkuje błędem kompilacji.

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



Rysunek 4:

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



Rysunek 5:

- 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