Zaawansowane programowanie obiektowe - wykład 2

dr Piotr Jastrzębski

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

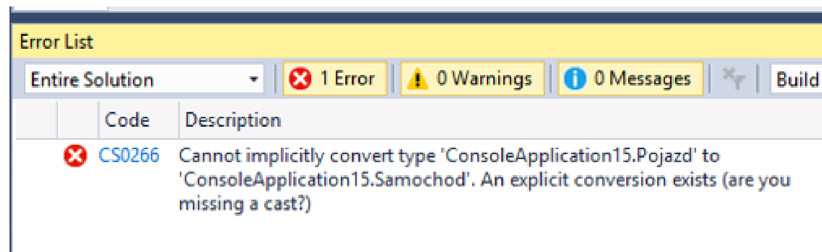
- Rzutowanie w górę (zawsze możliwe i skuteczne).
  - Samochód możemy traktować jak Pojazd.
- Rzutowania w dół (możliwe gdy rzeczywiście obiekt jest typu pochodnego).
  - Nie każdy Pojazd jest Samochodem.
  - Nie każdy prostokąt jest kwadratem.

Rzutowanie w górę

```java
//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

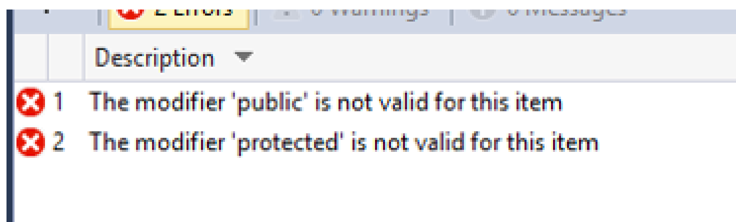
Interfejs – definicja abstrakcyjnego typu posiadającego jedynie operacje, a nie dane. Kiedy w konkretnej klasie zdefiniowane są wszystkie metody interfejsu mówimy, że klasa implementuje dany interfejs. W programie mogą być tworzone zmienne typu referencja do interfejsu, nie można natomiast tworzyć obiektów 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 implementacja 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
    }
}
```

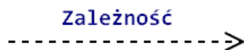
Przykłady implementacji systemowych interfejsów

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

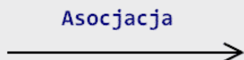
Temat klonowania, kopiowania a interfejs `ICloneable` - gist.

Związki pomiędzy klasami a UML

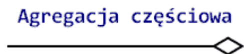
Związki pomiędzy klasami a UML



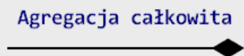
Gdy jedna klasa chwilowo wykorzystuje drugą, lub wie o jej istnieniu



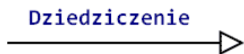
Gdy jedna klasa wykorzystuje drugą, ale nie są zależne



Gdy klasa zawiera drugą klasę, ale współdzieli odwołanie do niej z inną



Gdy klasa zawiera drugą klasę, i są od siebie zależne

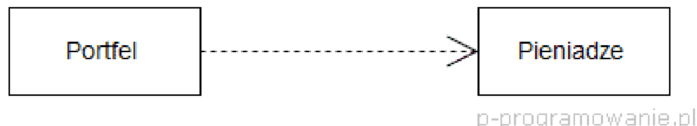


Gdy jedna klasa jest rozszerzeniem drugiej, i współdzielą swoje funkcjonalnośći

p-programowanie.pl

Zależność

Zależność – najsłabszy związek znaczeniowy między klasami, gdy jedna z nich używa innych klas. Na diagramie klas oznaczana przerywaną linią zakończoną strzałką wskazującą kierunek zależności.

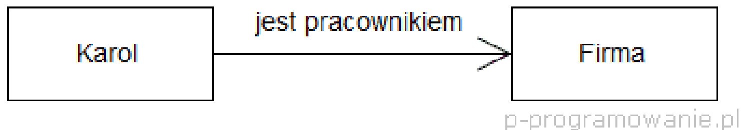


```
class Portfel
{
   void Dodaj(Pieniadze p)
   {

   }
}
```

Asocjacja

Asocjacja wskazuje na trwałe powiązanie pomiędzy obiektami danych klas (np. firma zatrudnia pracowników). Na diagramie asocjację oznacza się za pomocą linii, która może być zakończona strzałką (oznaczającą kierunek powiązania klas). Nazwę cechy wraz z krotnością umieszcza się w punkcie docelowym asocjacji.

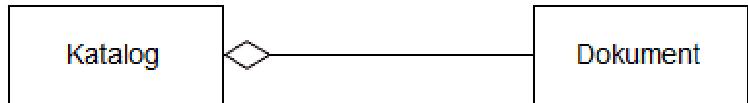


Agregacja

Agregacja (inaczej zawieranie się, gromadzenie) – sytuacja, w której tworzy się nową klasę, używając klas już istniejących (często nazywa się to "tworzeniem obiektu składowego"). Nowa klasa może być zbudowana z dowolnej liczby obiektów (obiekty te mogą być dowolnych typów) i w dowolnej kombinacji, by uzyskać żądany efekt. Agregacja jest często określana jako relacja typu "zawiera" np. "samochód zawiera silnik" - gdzie "samochód" i "silnik" są klasami, oraz klasa "samochód" zawiera w sobie obiekt (czasami referencję czy wskaźnik) typu "silnik".

Agregacja częściowa

Agregacja częściowa - sytuacja, w której element częściowy może należeć do elementu głównego, jednak nie jest od niego zależny. Usunięcie elementu głównego nie wpływa na usunięcie elementu częściowego. Element częściowy może także należeć do wielu elementów głównych.



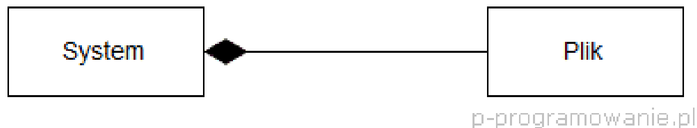
p-programowanie.pl

```
class Katalog
{
   private Dokument swiadectwo;

   public void SetSwiatectwo(Dokument swiadectwo)
   {
      swiadectwo = swiadectwo;
   }
}
```

Agregacja całkowita

Agregacja całkowita - sytuacja, w której po usunięciu klasy głównej, zostanie usunięta klasa częściowa.



```
class System
{
    private Plik _plik1 = new Plik();
}
```

```
class System
{
   private Plik plik1;

   public System()
   {
      plik1 = new Plik();
   }
}
```

Polimorfizm

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

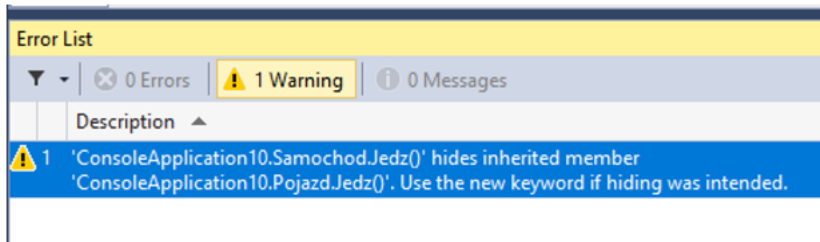
Napisanie metody - to nie jest polimorfizm

```
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`

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

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

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

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

```
abstract class Pojazd
{
    public abstract void Jedz();
}

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