Wyższa Szkoła Bankowa
Wydział Finansów i Zarządzania
Informatyka Inżynierska
Studia I stopnia, semestr 2
Gdańsk

Programowanie obiektowe
Laboratorium 5 & 6

Autor:
Jakub Głuszek

16 maja 2020

Spis treści

1 Lab 5 **3**
1.1 Dziedziczenie . . . 3
1.2 Polimorfizm . . . 3
1.3 Przeciążanie metod i konstruktorów . . . 4
1.4 Modyfikator dostępu protected . . . 4
1.5 Słowo kluczowe `base` . . . 4

2 Lab 6 **5**
2.1 Składowe wirtualne i abstrakcyjne . . . 5
2.2 Klasa abstrakcyjna . . . 5
2.3 Pieczętowanie funkcji . . . 6
2.4 Tworzenie interfejsów . . . 6

1 Lab 5

1.1 Dziedziczenie

Stwórz klasę `Animal`, która posłuży za klasę bazową dla dwóch innych klas pochodnych. Dodaj do niej kilka publicznych właściwości np. `Name, Height, Weight`.

Utwórz dwie klasy pochodne, które będą dziedziczyć po klasie bazowej `Animal`. Dla przykładu niech będą to klasy `Dog` oraz `Snake`. Aby utworzyć klasę pochodną w C# stosujemy symbol ":".

```
public class Dog : Animal
{

}
```

W klasie `Main` utwórz obiekt **jednej** z klas pochodnych (np. `Dog`) i sprawdź jakie ma dostępne właściwości (napisz nazwę obiektu, następnie kropkę i Ctrl+Space - powinni się wyświetlić dostępne składowe klasy).

Przypisz dowolne wartości tym składowym i wypisz je na ekranie konsoli za pomocą metody `WriteLine` typu `Console`.

Utwórz metodę `ShowInfo()` w klasie `Animal` i przenieś do niej napisany przed chwilą kod drukujący na ekranie konsoli podstawowe informacje. Wywołaj `ShowInfo()` w metodzie `Main` i sprawdź poprawność drukowania tych informacji.

Dodaj nową metodę do klasy `Dog` na przykład `WagItsTail`:

```
public void WagItsTail()
{
  Console.WriteLine("I'm wagging my tail...");
}
```

Wywołaj ją w klasie `Program`.

1.2 Polimorfizm

Zmienna jednego typu może odnosić się do obiektu typu, który jest podklasą tego typu - jest to jeden z rodzajów polimorfizmu[1]. Wynika to z faktu, że klasa pochodna posiada wszystkie składowe klasy bazowej. Tzn. że np. każdy pies jest w istocie zwierzęciem tak samo jak każdy wąż. Należy jednak zauważyć, że twierdzenie odwrotne nie jest prawdziwe. Dlatego możemy tworzyć obiekt typu `Dog` i przypisać referencję do niego do zmiennej typu `Animal`:

```
Animal myAnimal = new Dog();
```

W przypadku „odwrotnym" kompilator zgłosi nam błąd.

```
Dog myDog = new Animal(); //compile-time error
```

Utwórz obiekt typu `Dog`, przypisz referencję do niego do zmiennej typu `Animal` (tak jak zostało to pokazane powyżej) i sprawdź jakie posiada składowe. Dodatkowo upewnij się kompilator nie pozwoli Ci na utworzenie obiektu typu `Animal` i przypisanie go do zmiennej typu `Dog`. Zakomentuj tę linijkę (Ctrl+K+C).

Referencje do obiektu można niejawnie rzutować w górę (zawsze się udaje) albo jawnie w dół (udaje się, gdy obiekt ma odpowiedni typ). Sprawdź te dwie sytuacje, zakomentuj linijkę generującą błędy kompilacji.

```
Dog myDog = new Dog();
Animal a = myDog; //rzutowanie w gore

Console.WriteLine(a.Name);
a.WagItsTail(); //blad kompilacji

Dog d = (Dog)a;
a.WagItsTail(); //wywolanie metody sie uda
```

[1] Polimorfizm (wielopostaciowość) jest mechanizmem, który pozwala na różne zachowywanie się obiektów podczas wykonywania programu.

Jeżeli próba rzutowania w dół się nie uda, zostanie zgłoszony wyjątek `InvalidCastException` - jest to przykład działania kontroli typów podczas wykonywanie się programu.

Chętnym polecam dodatkowo zapoznać się i sprawdzić działanie operatorów `is`[2]oraz `as`[3].

1.3 Przeciążanie metod i konstruktorów

Innym rodzajem polimorfizmu (statycznym) jest przeciążanie metod, konstruktorów[4]czy operatorów[5]. Pozwala on na utworzenie w danej klasie wielu metod/konstruktorów o takiej samej nazwie. Typ może przeciążać metody pod warunkiem, że posiadają one inną sygnaturę.

Uwaga: Typ zwrotny (oraz modyfikator `params`) nie wchodzą w skład sygnatury.

Dodaj do klasy `Animal` dwie metody `Age` o takiej samej nazwie, ale zawierające inną liczbę/typy parametrów. Zaimplementuj brakującą metodę `SetItsAge()` i sprawdź poprawność przypisywania wartości do zmiennej `Age` w metodzie `Main()`. Dodatkowo możesz ustawić akcesor `set` tej właściwości na prywatny.

```
public int Age { get; set; }

public void SetItsAge(int age)
{
  Age = age;
}

public void SetItsAge(DateTime BirthDate)
{
  //...
}
```

1.4 Modyfikator dostępu protected

Składowa klasy oznaczona jako `protected` jest dostępna w klasie pochodnej jednak nie jest widoczna „na zewnątrz" obiektu. Jeżeli oznaczymy nasze właściwości w klasie `Animal` jako `protected` w metodzie `Main()` nie będzie do nich dostępu.

Sprawdź działanie modyfikatora `protected`. Utwórz dwie dodatkowe, dowolne właściwości i oznacz jedną z nich modyfikatorem dostępu `protected`. Upewnij się dostęp do niego istnieje jedynie w klasie bazowej i w klasach pochodnych - nie ma możliwości odczytania jej poza klasą. Modyfikatory można dodawać również do pojedynczych akcesorów np. `public MyProperty {get; protected set;}`, oznacz akcesor drugiej z przed chwilą utworzonych właściwości:

```
protected string Order { get; set; }
public string Family { get; protected set; }
```

1.5 Słowo kluczowe `base`

Jeżeli klasa bazowa posiada konstruktor z parametrami konieczne jest przekazanie do niego tych parametrów w klasie pochodnej. Podczas tworzenia klasy pochodnej jest wykonywany konstruktor klasy bazowej więc muszą zostać do niego przekazane odpowiednie parametry. W tym celu używa się słowa kluczowego `base`. Kolejność wywoływania konstruktorów prześledź „pod" debuggerem.

Dodaj do klasy bazowej `Animal` konstruktor, w którym będzie przypisywana wartość do właściwości `Name` (ustaw akcesor `set` jako prywatny). Dodatkowo utwórz nową właściwość w klasie `Dog` określającą rasę i również przypisz jej wartość w konstruktorze klasy `Dog` tak jak zostało to pokazane poniżej:

[2] Operator `as` wykonuje rzutowanie w dół, ale w razie niepowodzenia operacji zamiast zgłaszać wyjątek, zwraca wartość null.

[3] Operator `is` sprawdza czy obiekt jest pochodną pewnej klasy. Wykorzystuje się go często przed wykonaniem rzutowania w dół.

[4] Przeciążanie operatorów pozwala na zapewnienie bardziej naturalnej składni dla typów własnych programisty. O przeciążaniu operatora `==` wspominaliśmy na poprzednich zajęciach. Analogicznie można przeciążać inne operatory.

[5] Przeciążanie konstruktorów odbywa się tak samo jak przeciążanie metod. Dodatkowo jeżeli chcemy uniknąć powielania kodu (np. część isntrukcji w obu konstruktorach się powtarza) możemy wykorzystać słowo kluczowe `this` (umieszczone po nazwie konstruktora i symbolu „:") np. `public Dog(string name, string breed, Color fur):this(name, breed)`. W takiej sytuacji najpierw zostanie wykonany kod konstruktora z argumentami `name` oraz `breed`, następnie kod konstruktora z parametrami `name`, `breed` i `fur` (jeżeli klasa dodatkowo jest pochodną innej klasy, to na samym początku zostanie wykonany kod konstruktora klasy bazowej).

```
public string Breed {get;private set;}
public Dog(string name, string breed) : base(name)
{
  Breed = breed;
}
```

Analogiczną zmianę wykonaj w klasie `Snake`, możesz pominąć tworzenie właściwości `Breed`.

2 Lab 6

2.1 Składowe wirtualne i abstrakcyjne

Do składowej (metody, właściwości, indeksatora lub deklaracji zdarzenia) możemy dodać słowo kluczowe `virtual` albo `asbtract`. Składowa wirtualna posiada implementację w klasie bazowej, a klasa pochodna **może** ją wykorzystać albo za pomocą operatora `override` ja przysłonić (zaproponować własną implementację). Natomiast składowa abstrakcyjna nie posiada implementacji w klasie bazowej, a klasa pochodna **musi** ją przysłonić.

Dodaj do klasy `Animal` metodę wirtualną `SayHello`:

```
public virtual void SayHello()
{
  Console.WriteLine($"I'm an animal and my name is {Name}");
}
```

W klasie pochodnej `Dog` przesłoń tę metodę (np. w celu zaproponowania lepszej/bardziej szczegółowej implementacji) w następujący sposób:

```
public override void SayHello()
{
  Console.WriteLine($"I'm a DOG and my name is {Name} - Woof! Woof!");
}
```

W klasie `Snake` nie przesłaniaj ww. metody. Utwórz w metodzie `Main()` trzy obiekty: zwierzęcia, psa oraz węża i dla każdego z nich wywołaj metodę `SayHello()`.

Innym bardziej życiowym przykładem wykorzystania metody wirtualnej może być np. przesłonięcie pewnego standardowego algorytmu innym bardziej dokładnym w klasie pochodnej.

2.2 Klasa abstrakcyjna

Słowem kluczowym `asbtract` oznaczamy całą klasę i jej składowe. Jeżeli klasa jest abstrakcyjna nie ma możliwości stworzenia jej instancji. Możliwe jest jedynie tworzenie instancji klas, które po niej dziedziczą (chyba, że również będą abstrakcyjne). Oznacz klasę `Animal` jako abstrakcyjną, co się zmieniło w metodzie `Main()`, czy kompilator pozwala na utworzenie instancji klasy `Animal`. **Zakomentuj** te fragmenty kodu, które sprawiają problemy. Następnie dodaj do klasy `Animal` **metodę** abstrakcyjną `Cry()`.

Metodę abstrakcyjną można traktować jako metodę, która nie została skończona. Jeżeli tak się dzieję to również cała klasa jest „nie skończona" i dlatego musi zostać oznaczona jako abstrakcyjna. Nie ma możliwości oznaczenia metody słowem kluczowym `abstract` w nieabstrakcyjnej klasie.

```
public abstract class Animal(string name)
{
  //...
  public abstract void Cry();
  //...
}
```

Ponieważ klasy `Dog` oraz `Snake` nie są abstrakcyjne **muszą** one posiadać/implementować nie abstrakcyjną metodę `Cry()`, która zgodnie z sygnaturą metody w klasie bazowej nie będzie przyjmować żadnych argumentów oraz będzie typu zwrotnego `void`. Jeśli metoda ta nie zostanie zaimplementowana, program „nie przejdzie" kompilacji.

Zaimplementuj tę metodę w **dwóch** klasach pochodnych i wywołaj je z klasie `Program`.

```
public override void Cry()
{
  Console.WriteLine("Woof! Woof! Woof!");
}
```

2.3 Pieczętowanie funkcji

Jeżeli chcemy uniemożliwić klasie dalsze przesłanianie metod przez jej kolejne podklasy możemy ją zapieczętować[6] posługując się operatorem `sealed`. Sprawdź działanie słowa kluczowego `sealed`. Zapieczętuj metodę abstrakcyjną `Cry()` w klasie `Snake` i spróbuj przysłonić ją w nowej klasie np. `Cobra` (dodaj ją do projektu - niech dziedziczy po klasie `Snake`). Zwróć uwagę, że jest to nie możliwe. Jeżeli usunelibyśmy operator `sealed`, operacja przesłonięcia byłaby dostępna (nie usuwaj go, nie nadpisuj metody `Cry()` w typie `Cobra`).

2.4 Tworzenie interfejsów

Interfejs jest podobny do klasy abstrakcyjnej z tym wyjątkiem, że wszystkie jego składowe są abstrakcyjne - nie posiada on implementacji żadnej ze składowych. Opisuje on zachowanie danej klasy. Klasa może implementować wiele interfejsów (w C# dziedziczyć klasa może tylko po jednej klasie bazowej[7]).

Deklaracja interfejsu jest podobna do deklaracji klasy. Aby stworzyć interfejs wykorzystujemy słowo kluczowe `interface` i dalej nazwę interfejsu (tak jak klasę, interfejs umieszczamy w osobnym pliku) zazwyczaj z literą `I` na początku nazwy.

Stwórz w nowym pliku interfejs `IPrintable`:

```
interface IPrintable
{
  void Print();
}
```

Klasa, która implementuje interfejs musi zaimplementować **wszystkie** jego składowe. Dodaj do projektu (również w osobnym pliku) klasę `Printer`, która będzie implementowała stworzony przed chwilą interfejs:

```
class Printer : IPrintable
{
  public void Print()
  {
    Console.WriteLine("I'm a printer and I'm printin'...");
  }
}
```

Stwórz kolejny interfejs `ICopiable` i umieść w nim deklarację metody `Copy` - analogicznie jak w interfejsie `IPrintable`. W projekcie utwórz klasę kserokopiarki, która będzie implementowała oba utworzone wcześniej interfejsy (kolejne interfejsy oddzielamy przecinkami):

```
class Photocopier : IPrintable, ICopiable
{
  //...
}
```

Utwórz dwie referencje do interfejsów (nie można tworzyć obiektów tego typu) tak jak zostało to pokazano poniżej i sprawdź jakie posiadają składowe:

```
IPrintable printer = new Printer();
ICopiable copier = new Photocopier();
```

Powinniśmy utrzymywać interfejsy tak „małe" jak to tylko możliwe. Dlatego stworzyliśmy dodatkowy interfejs `ICopiable` i zaimplementowaliśmy go w klasie `Photocopier` obok `IPrintable`. Złym rozwiązaniem byłoby zbudowanie interfejsu `IPrintableAndCopiable`. Takie duże interfejsy są nazywane *fat* albo *polluted*.

[6] Zapieczętowana może zostać również cała klasa.

[7] Część języków programowania wspiera możliwość wielodziedziczenia (ang. multiple inhertance). Przykładowymi językami posiadającymi ten mechanizm są C++, Python czy R.

Klient powinien implementować jedynie te interfejsy, które są mu potrzebne. Jeśli są one małe i lekkie minimalizujemy ryzyko, że któreś ze składowych tych interfejsów nie będą potrzebne.

Załóżmy teraz, że tworzysz klasę która będzie odpowiedzialna za drukowanie listy wykorzystanych elementów np. do wykonania obwodu drukowanego. Nazwijmy tę klasę `PcbBom`. Dodaj ją do projektu i utwórz w niej prywatne pole typu `IPrintable`. Dodatkowo stwórz konstruktor, który będzie przyjmował jako parametr obiekt implementujący interfejs `IPrintable`. W konstruktorze przypisz przekazywany obiekt do pola `printer`. Niech klasa posiada również publiczną metodę `Print()`, która będzie wywoływała metodę `Print()` obiektu (implementującego ten interfejs) przekazanego w konstruktorze.

```
class PcbBom
{
  IPrintable printer;

  public PcbBom(IPrintable somethingThatPrints)
  {
    //...
  }

  public void Print()
  {
    //...
  }
}
```

Zwróć uwagę, że w klasie `PcbBom` wykorzystując obiekt implementujący `IPrintable` możemy korzystać tylko ze składowych, które znajdują się w deklaracji tego interfejsu.

Utwórz teraz w metodzie `Main` dwa obiekty typu `PcbBom`. Do jednego przekaż obiekt typu `Printer`, a drugiego `Photocopier`. Dwukrotnie wywołaj metodę `Print()` i zobacz, że w obu przypadkach do wydrukowania listy materiałów został wykorzystany inny obiekt.

```
  IPrintable printer = new Printer();
  IPrintable photocopier = ...


  var pcbBomA = new PcbBom(...);
  ...

  pcbBomA.Print();
  pcbBomB.Print();
```

Jeżeli byśmy chcieli drukować tę listę za pomocą jeszcze innego urządzenia wystarczy stworzyć nową klasę, która będzie implementowała interfejs `IPrintable`. Żadne dodatkowe modyfikacje w klasie `PcbBom` nie będą już potrzebne. Takie składanie obiektu z innych obiektów nazywamy kompozycją[8](albo agregacją[9]). Zazwyczaj warto jest przekładać kompozycję nad dziedziczenie[10].

[8] Kompozycja jest podobna do agregacji. Z tą różnicą, że obiekty nie mogą istnieć niezależnie od siebie (jest to silniejsza forma połączenia dwóch obiektów). Np. pokoje nie mogą istnieć bez domu.

[9] Agregacja jest komponowaniem obiektu z innych istniejących obiektów (jeden obiekt „posiada" drugi). W przypadku agregacji oba obiekty mogą istnieć niezależnie od siebie. Np. obiekt typu student, może istnieć bez obiektu uczelni.

[10] Istnieje zasada w programaniu obiektowym, która mówi, że klasy powinny uzyskiwać polimorfizm poprzez kompozycję, a nie przez dziedziczenie. Pozwala ona na uzyskanie większej elastyczności, łatwiejszego rozwijania i modyfikowania kodu.

Bibliografia

[1] Ben Albahari Joseph Albahari. C# 7.0 w pigułce. Helion, Gliwice 2018.