

Michał Włodarczyk

Programowanie obiektowe

ITA - 105

Wersja 2

© 2009 Michał Włodarczyk. Autor udziela prawa do bezpłatnego kopiowania i dystrybuowania wśród pracowników uczelni oraz studentów objętych programem MSDNAA.

Wszelkie informacje dotyczące programu można uzyskać:
pledu@microsoft.com.

Wszystkie inne nazwy firm i producentów wymienione w niniejszym dokumencie mogą być znakami towarowymi zarejestrowanymi przez ich właścicieli.

Inne produkty i nazwy firm używane w treści mogą być nazwami zastrzeżonymi przez ich właścicieli.

Opis kursu

Kurs przeznaczony jest do prowadzenia przedmiotu programowanie obiektowe na uczelniach wyższych. Skupia się na nauce programowania obiektowego w języku C# w środowisku programistycznym Visual Studio 2008. Kurs prowadzi przez możliwości programowania zorientowanego obiektowo spotykane w wielu językach.

Kurs składa się z 13 modułów. Każdy moduł zawiera wprowadzenie w tematykę wraz z odpowiednimi przykładami, ułatwiającymi zrozumienie omawianych zagadnień, podsumowanie i pytania sprawdzające zdobytą wiedzę. Moduły zawierają również zadania laboratoryjne przeznaczone do wykonania przez studentów w pozostałym czasie zajęć lub w wolnym czasie w celu utrwalenia zdobytych umiejętności.

Cel kursu

Po zrealizowaniu kursu – przyswojeniu prezentowanej wiedzy oraz utrwaleniu umiejętności przez wykonanie ćwiczeń laboratoryjnych, student będzie potrafił tworzyć programy według paradygmatu programowania zorientowanego obiektowo. Słuchacze będą znali pojęcia związane z programowaniem obiektowym, nauczą się tworzyć kod w języku C# realizujący mechanizmy programowania obiektowego, będą też w stanie rozwiązywać problemy za pomocą tych mechanizmów.

Uczestnicy kursu zostaną również przygotowani do kursów z metodologii, projektowania i inżynierii oprogramowania, dzięki wprowadzeniu elementów notacji UML oraz wzorców projektowych.

Wymagania wstępne

Wymagania wstępne, które muszą spełniać słuchacze tego kursu, obejmują:

- wskazana znajomość języka programowania o składni podobnej do C#, jak na przykład C/C++, Pascal, Java, Perl,
- umiejętność programowania strukturalnego w dowolnym języku,
- umiejętność pracy w środowisku programistycznym Visual Studio

Opis modułów

Numer modułu Tytuł	Opis
Moduł 1 Pojęcie klasy	W tym module zapoznasz się z podstawowymi pojęciami z programowania obiektowego, takie jak klasa, obiekt, abstrakcja czy hermetyzacja. Dowiesz się, jak w języku C# definiujemy klasę, również jak definiuje się klasy częściowe. Nauczysz się również sterować dostępem do części składowych klasy przy pomocy słów public i private. Poznasz znaczenie słowa this.
Moduł 2 Konstruktor	W tym module zapoznasz się z pojęciem konstruktora oraz nauczysz się, jak definiować go w języku C#. Omówiona zostanie również lista inicjalizacyjna konstruktora. Poznasz, co to są inicjalizatory obiektów i jak ich używać.

	Dowiesz się również co to jest destruktor. Zostanie również wprowadzone pojęcie wzorców projektowych i przedstawiony wzorec projektowy o nazwie prototyp.
Moduł 3 Właściwości i indeksatory	W tym module zapoznasz się z pojęciami właściwości i indeksatora oraz nauczysz się nimi posługiwać w języku C#. Poznasz też co to są właściwości automatyczne. Dodatkowo nauczysz się, co to są pola tylko do odczytu i jak je definiować. Zostanie tu również przedstawiony wzorec projektowy o nazwie proxy i jak przy jego pomocy uniezależnić definicję klasy od biblioteki wejścia-wyjścia.
Moduł 4 Składowe statyczne	W tym module zapoznasz się z pojęciem składowych statycznych. Dowiesz się do czego służą i jak się korzysta w języku C# z pól i metod statycznych. Nauczysz się co to jest konstruktor statyczny oraz po co go się definiuje. Dowiesz się również co to są klasy statyczne oraz co to są metody rozszerzające i jak je implementujemy. Zostanie tu również przedstawiony wzorec projektowy o nazwie Singleton.
Moduł 5 Przeciążenie operatorów	W tym module zapoznasz się z pojęciami przeciążenia operatorów. Dowiesz się, które operatory można przeciążyć w języku C#. Nauczysz się jak definiować metody, które będą implementowały żądany operator. Poznasz szczegóły dotyczące implementacji poszczególnych operatorów. Dowiesz się jak można zdefiniować w języku C# metody, które będą zamieniać obiekt jednego typu na obiekt innego typu.
Moduł 6 Dziedziczenie	W tym module zapoznasz się z jednym z podstawowych pojęć programowania obiektowego, a mianowicie dziedziczeniem. Nauczysz się jak dziedziczenie można wykorzystać w języku C#. Poznasz też pozostałe modyfikatory dostępu (protected, internal, internal protected). Dowiesz się również, jak inicjalizować obiekty klas pochodnych oraz jak przesłonić metodę klasy bazowej.
Moduł 8 Polimorfizm i funkcje wirtualne	W tym module zapoznasz się z kolejnym z głównych pojęć programowania obiektowego, a mianowicie polimorfizmem. Nauczysz się jak definiować metody wirtualnych. Dowiesz się jakie metody typu object można nadpisać i jak to zrobić. Poznasz kolejne wzorce projektowe o nazwie metoda fabrykująca i adapter.
Moduł 9 Klasy abstrakcyjne i interfejsy	W tym module zapoznasz się z pojęciem klas i metod abstrakcyjnych oraz uzyskasz informacje jak je implementować w języku C#. Dowiesz się również co to jest interfejs i do czego służy. Poznasz również pojęcie klas i metod zamkniętych. Zobaczysz również, jak można w

	języku C# sprawdzić, czy zmienna klasy bazowej, zawiera referencję do obiektu, którego typ jest żadaną klasę pochodną. Poznasz również wzorzec projektowy fabryka abstrakcyjna.
Moduł 10 Metody i typy generyczne	W tym module zapoznasz się z pojęciem metod i typów generycznych. Dowiesz się do czego służą i jak można je definiować w języku C#. Zobaczysz jakie ograniczenia można stosować do typów parametryzujących metodę lub typ ogólny jak również po co stosować te ograniczenia. Poznasz również kilka typów generycznych reprezentujących podstawowe struktury danych.
Moduł 11 Implementacja przykładowych interfejsów	W tym module zapoznasz się z podstawowymi interfejsami z których można skorzystać tworząc programy na platformę .Net.: IDisposable, IEnumerable, IEnumerator, IComparable i Comparer. Nauczysz się również jak można je zaimplementować we własnych klasach. Dowiesz się również do czego służy blok instrukcji using. Poznasz składnię iteratorów przy pomocy których w łatwy sposób można zaimplementować wzorzec projektowy iterator.
Moduł 12 Delegacje i zdarzenia	W tym module zapoznasz się z pojęciami delegacji i zdarzenia. Dowiesz się co to jest delegacja i jak je można definiować, zarówno w standardowy sposób jak i przy pomocy metod anonimowych i wyrażeń lambda. Nauczysz się również definiować własne zdarzenia. Poznasz również wzorzec projektowy Obserwtor.
Moduł 13 Refleksja i atrybuty	W tym module zapoznasz się z pojęciem refleksji, czyli mechanizmie uzyskania informacji o typie w czasie działania programu. Poznasz jak z mechanizmu refleksji można skorzystać w języku C#. Dowiesz się też jak dodać dodatkowe informacje (metadane) do programu przy pomocy atrybutów. Nauczysz się też definiować własne atrybuty (klasy reprezentujące atrybuty) i je stosować.
Moduł 14 Serializacja	W tym module zapoznasz się z pojęciem serializacji. Poznasz rodzaje serializacji wspieranych przez .Net Framework oraz nauczysz się jak tworzyć klasy serializowalne i ich używać. Zobaczysz również jak można dostosować proces serializacji, aby spełniał Twoje oczekiwania. W module opisana jest serializacja binarna oraz serializacja XML.

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 1

Wersja 2

Pojęcie klasy

Spis treści

Pojęcie klasy.....	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	9
Porady praktyczne	13
Uwagi dla studenta	14
Dodatkowe źródła informacji	15
Laboratorium podstawowe	16
Problem 1 (czas realizacji 30 minut)	16
Problem 2 (czas realizacji 15 minut)	18
Laboratorium rozszerzone	22
Zadanie 1 (czas realizacji 90 min)	22
Zadanie 2 (czas realizacji 30 min)	22

Informacje o module

Opis modułu

W tym module zapoznasz się z podstawowymi pojęciami z programowania obiektowego, takimi jak klasa, obiekt, abstrakcja czy hermetyzacja. Dowiesz się, jak w języku C# definiujemy klasę, również jak definiuje się klasy częściowe. Nauczysz się także sterować dostępem do części składowych klasy przy pomocy słów `public` i `private`. Poznasz znaczenie słowa `this`.

Cel modułu

Celem modułu jest pokazanie, jak w języku C# można tworzyć klasy oraz sterować dostępem do ich składowych.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- znał podstawowe pojęcia związane z programowaniem obiektowym
- potrafił tworzyć własne klasy i z nich korzystać
- potrafił sterować dostępem do składowych klasy
- umiał generować diagram klas w środowisku Visual Studio

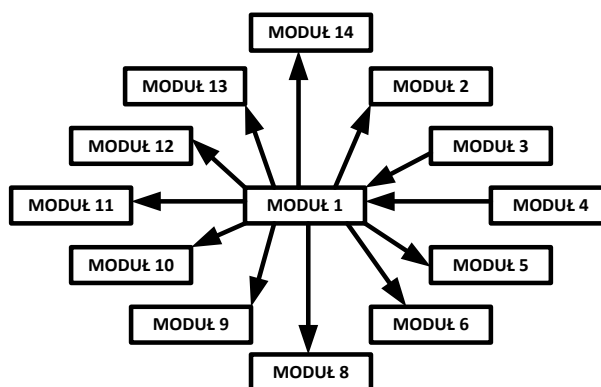
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinienś:

- wiedzieć, co to jest Platforma .NET
- wiedzieć, jaka jest struktura programu w języku C#
- znać różnice między typem referencyjnym a typem bezpośrednim
- umieć korzystać ze środowiska Visual Studio
- znać podstawy programowania proceduralnego (instrukcje sterujące, funkcje, tablice, struktury, obsługa wyjątków)
- korzystać z plików w języku C#

Mapa zależności modułu

Przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym kursie „Wprowadzenie do programowania” lub posiadać odpowiednią wiedzę.



Rys. 1 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Jesteś programistą z pewnym bagażem doświadczeń. Twoje programy stają się coraz większe i zaczynasz mieć problemy z ich opanowaniem. Zadajesz sobie pytanie, czy jest możliwe podzielenie programu na oddzielne jednostki, które możesz traktować jako samodzielne byty, współdziałające ze sobą, aby zrealizować określone zadanie i które możesz wielokrotnie używać, oddzielnie testować. Pewnie zastanawia Cię też, czy możesz zabezpieczyć swoje struktury tak, aby nie można było w nich dokonać nieodpowiednich modyfikacji, prowadzących do zepsucia egzemplarza twojej struktury. Modyfikacja jednego pola może pociągać za sobą konieczność modyfikacji innego. Chcesz tak projektować struktury, żeby można było z nich łatwo i bezpiecznie korzystać, bez nieustannego pamiętania, które pola są ze sobą powiązane. Programista też jest człowiekiem i jeżeli da mu się okazję do popełnienia błędu, to pewnie błąd ten popełni.

Twoje struktury powinny przypominać rzeczy, którymi posługujesz się na co dzień. Włączając telewizor lub radio nie boisz się, że możesz je popsuć. To, co jest niebezpieczne i może prowadzić zarówno do uszkodzenia telewizora, jak i jego użytkownika, zamknięte jest w hermetycznej obudowie. Wreszcie, czy pisząc program nie możesz patrzeć na niego w sposób zbliżony do rzeczywistego problemu, a nie jako zbiór parametrów wejściowych, który należy przekształcić w określony zbiór parametrów wyjściowych? Tworząc postać w jakiejś grze raczej nie myślisz o niej jako bycie mającym tylko pewne parametry. Twoja postać potrafi biegać, prawdopodobnie również strzelać, czyli oprócz parametrów ją charakteryzujących ma również specyficzne zachowanie. Czy tak można programować? Oczywiście że tak – stosując programowanie obiektowe.

Podstawy teoretyczne

Programowanie obiektowe lub *programowanie zorientowane obiektowo* (ang. *object-oriented programming*) jest jednym z paradygmatów programowania, czyli pewnym sposobem patrzenia na tworzenie oprogramowania.

Istotnym atutem programowania obiektowego, dzięki któremu jest ono tak popularne, jest traktowanie programów komputerowych w bardzo podobny sposób do otaczającej nas rzeczywistości. Każdy element otaczającej rzeczywistości klasyfikujemy, czyli tworzymy grupy elementów, które posiadają te same właściwości (parametry) i podobne zachowanie (funkcjonalność). Dzięki temu elementom tym można nadać nazwę. Mówiąc słowo „samochód” wyobrażamy sobie coś, co posiada markę, kolor, silnik (są to parametry) oraz potrafi poruszać się, może przyspieszyć, można w nim włączyć wycieraczki, zmienić bieg (funkcjonalność). W programowaniu obiektowym zbiór elementów, które posiadają ten sam zbiór parametrów oraz tę samą funkcjonalność nazywamy *klasą*, natomiast konkretny element, egzemplarz danej klasy nazywamy *obiekt*em. Klasa jest niejako szablonem, według którego budowane są obiekty. Zbiór wartości wszystkich parametrów obiektu nazywamy *stanem obiektu*. Klasą więc jest samochód, zaś obiektem konkretny samochód, np. ten, którym przyjechałeś na zajęcia. W przypadku programowania obiektowego zmienia się definicja programu komputerowego. Program komputerowy nie definiujemy jako ciąg instrukcji, który ma być wykonany, ale jako zbiór obiektów powiązanych ze sobą różnymi zależnościami i przekazujących sobie wzajemnie informacje o tym, co należy zrobić za pomocą pewnych sygnałów zwanych komunikatami.

U podstaw programowania zorientowanego obiektowo leżą następujące podstawowe pojęcia:

- abstrakcja
- hermetyzacja lub enkapsulacja (ang. *encapsulation*)
- dziedziczenie (ang. *inheritance*)
- polimorfizm (ang. *polymorphism*)

Abstrakcja jest związana z każdym modelowaniem rzeczywistości. Polega ono na tym, że uwzględniamy tylko te parametry i tę funkcjonalność, które są istotne ze względu na rozwiązywany problem, a pomijane są pozostałe.

Hermetyzacja polega na tym, że zbiór parametrów i funkcjonalność są ze sobą połączone w postaci pojedynczej jednostki zwanej oczywiście obiektem. Z hermetyzacją związane jest pojęcie ukrywania informacji, które polega na tym, że pewne parametry i pewna funkcjonalność nie jest ogólnie dostępna. Więcej na ten temat będzie można się dowiedzieć w dalszej części tego modułu.

Dziedziczenie polega na tworzeniu pewnego związku między klasami. Związek ten określa, że pewna klasa jest szczególnym przypadkiem innej klasy, np. pojazd i samochód, czy zwierzę i lew. Więcej na ten temat można przeczytać w module 6.

Polimorfizm polega na tym, że decyzja, która funkcja (metoda) ma być wykonana, jest podejmowana w czasie działania programu, a nie w czasie kompilacji. Więcej na ten temat można się dowiedzieć w module 8.

Definicja klasy w języku C#

W języku C# klasę definiujemy przy pomocy słowa `class`, po którym następuje nazwa klasy. Definiując klasę budujesz nowy typ, a co za tym idzie, dajesz programiście do dyspozycji nowe pojęcie, którego może on używać przy budowanie swojej aplikacji. Zasady nadawania nazw, jak i konwencje nazewnictwa, możesz znaleźć w kursie „Wprowadzenie do programowania”. Samą definicję klasy umieszczasz wewnątrz nawiasów klamrowych.

```
class nazwa_klasy {  
    definicja_składowych  
}
```

Słowo `class` może być poprzedzone *modyfikatorem dostępu*, o czym będzie szerzej mowa w module 6.

Czytaliśmy wcześniej w tym module, że klasę definiuje się przez zbiór parametrów oraz pewną funkcjonalność. Parametry definiujące stan obiektu, czyli egzemplarza danej klasy, nazywamy *polami* lub *zmiennymi składowymi* danej klasy. Pola definiujemy w następujący sposób:

```
[modyfikator_dostępu] typ_pola nazwa_pola;
```

Polu możesz od razu nadać wartość przez umieszczenie po nazwie pola znaku równości, a następnie wyrażenia, którego typ jest zgodny z typem pola:

```
[modyfikator_dostępu] typ_pola nazwa_pola = wyrażenie;
```

Szczegóły na temat inicjalizacji poszczególnych pól zostaną opisane w module 2.

Funkcjonalność klasy definiujemy przy pomocy metod, czyli funkcji zdefiniowanych wewnątrz klasy (funkcji składowej), w następujący sposób (zwróćmy uwagę na brak słowa kluczowego `static`):

```
[modyfikator_dostępu] typ_zwracany nazwa_metody(lista_parametrów) {  
    ciało_metody  
}
```

O modyfikatorze dostępu będzie można przeczytać w dalszej części tego modułu..

Pola i metody nazywamy *składowymi klasy*.

Mając klasę zdefiniowaną w następujący sposób:

```
class Klasa1 {  
    public int X;  
    public void Metoda1() {  
        Console.WriteLine("X = {0}", X);  
    }  
}
```

```
    }  
}
```

w celu utworzenia obiektu, najpierw tworzymy zmienną typu `Klasa1`, a następnie przypisujemy ją do obiektu utworzonego przy pomocy operatora `new`:

```
Klasa1 zmienna1;  
zmienna1 = new Klasa1();
```

Możemy to oczywiście zapisać w jednej linii:

```
Klasa1 zmienna1 = new Klasa1();
```

Powyższy kod pokazuje, że nazwa `zmienna1` nie symbolizuje obiektu, tylko zmienną, która może być powiązana z obiektem. Często będziemy jednak stosować skrót myślowy i obiekt będziemy nazywać nazwą zmiennej, która jest z nim połączona.

Do składowych obiektu możemy odwoływać się przy pomocy operatora kropki.

```
zmienna1.X = 20;           //odwołanie się do pola  
zmienna1.Metoda1();        //wywołanie metody
```

Słowo kluczowe `this`

Tworząc nowy obiekt, przydzielamy mu na zarządzanej stercie obszar pamięci, gdzie przechowywane są jego pola. Każdy obiekt ma własną „kopię” pól, kod metod natomiast jest przechowywany tylko raz w pamięci. Rozważmy klasę zdefiniowaną wcześniej oraz następujący kod:

```
Klasa1 a = new Klasa1();  
Klasa1 b = new Klasa1();  
a.X = 10;  
b.X = 20;  
a.Metoda1();    // Zostanie wypisane X = 10  
b.Metoda1();    // Zostanie wypisane X = 20
```

Zastanówmy się, jak to się dzieje, że następuje powiązanie między obiektem a metodą, która na rzecz tego obiektu jest wywołana. Gdy definiujemy metodę bez słowa kluczowego `static`, to niejawnie przyjmuje ona jako pierwszy argument zmienną o nazwie `this`. Typem argumentu jest klasa, w której dana metoda jest zdefiniowana, zaś argument jest przesyłany przez referencję, czyli definicję klasy:

```
class Klasa1 {  
    public int X;  
    public void Metoda1() {  
        Console.WriteLine("X = {0}", X);  
    }  
}
```

można sobie wyobrazić w następujący sposób (tylko teoretycznie):

```
class Klasa1 {  
    public int X;  
    public static void Metoda1(ref Klasa1 this) {  
        Console.WriteLine("X = {0}", this.X);  
    }  
}
```

Powyższy kod nie jest poprawny i nie skompiluje się, ponieważ `this` jest słowem kluczowym, więc nie wolno nadawać takiego identyfikatora zmiennym. Służy on tylko wytłumaczeniu znaczenia słowa `this`.

Definiowanie metod ze słowem kluczowym `static` zostało omówione w kursie „Wprowadzenie do programowania”, natomiast znaczenie słowa kluczowego `static` w programowaniu obiektowym zostanie przedstawione w module 4.

Wywołanie metody:

```
Klasa1 a = new Klasa1();  
a.Metoda1();
```

można sobie przetłumaczyć w następujący sposób:

```
Klasa1 a = new Klasa1();  
Klasa1.Metoda1(ref a);
```

Podsumowując, jeżeli definiujemy metodę i wewnątrz niej odwołujemy się do pól lub wywołujemy inną metodę danej klasy, to kompilator w czasie kompilacji każde odwołanie potraktuje jakby było poprzedzone słowem `this` ze znakiem kropki. Sami również możemy stosować taką formę odwołania do pól lub metod klasy. Przydaje się to zwłaszcza wtedy, gdy nazwa pola lub metody zostanie przesłonięta przez nazwę zmiennej lokalnej metody, co demonstruje poniższy przykład:

```
class Klasa2 {  
    public double X;  
    public static void Metoda1(double X) {  
        Console.WriteLine(X);           //odwołanie się do zmiennej lokalnej  
        Console.WriteLine(this.X);      //odwołanie się do pola klasy  
    }  
}
```

Ukrywanie informacji

Projektując swoją klasę możemy potrzebować nieraz dodatkowych pól lub metod, które nie są istotne z punktu widzenia działania naszej klasy lub można się do nich odwołać tylko w określony sposób, żeby nie zepsuć obiektu danej klasy. Weźmy np. pod uwagę telewizor. Konstruktorzy telewizora dali nam zestaw przycisków czy pokręteł, za pomocą których możemy sterować telewizorem. Telewizor składa się z wielu części, które są zamknięte w obudowie. Zamknięcie części w obudowie i udostępnienie na zewnątrz tylko tej funkcjonalności, która jest potrzebna z punktu widzenia użytkownika, ma następujące zalety:

- zapobiega zepsuciu telewizora
- ułatwia korzystanie z telewizora

Projektując klasę powinniśmy zwrócić uwagę, aby można było łatwo z niej korzystać i nie można było wprowadzić obiektu w stan, który byłby niedozwolony. Do zabezpieczenia składowych klasy przed nieodpowiednim dostępem służą modyfikatory dostępu. W języku C# wyróżniamy następujące modyfikatory dostępu:

- `private`
- `public`
- `protected`
- `internal`
- `internal protected`

Teraz poznamy tylko dwa pierwsze, pozostałe są opisane w module 6.

Modyfikator `public` oznacza, że dana składowa jest ogólnie dostępna i można się do niej odwoływać z metod innych klas. Składowe prywatne są to takie składowe, do których można się odwoływać tylko z metod klasy, w której dana składowa została zdefiniowana. Z metody danej klasy można również odwoływać się do składowych prywatnych innych obiektów tej samej klasy. Ilustruje to poniższy przykład:

```
class Klasa1 {
    public int X;
    private int y;
    public void Metoda1(Klasa1 k1) {
        X = 10;           //OK, odwołanie do "swojego" pola publicznego
        y = 15;           //OK, odwołanie do "swojego" pola prywatnego
        metoda1();        //OK, wywołanie "swojej" prywatnej metody
        k1.X = 10;        //OK, odwołanie do publicznego pola "własnej" klasy
        k1.y = 15;        //OK, odwołanie do prywatnego pola "własnej" klasy
        k1.metoda2();     //OK, wywołanie prywatnej metody "własnej" klasy
    }
    private void metoda2() {
    }
}

class Klasa2 {
    private double z;
    public void Metoda(Klasa1 k1) {
        k1.X = 33;        //OK, odwołanie do pola publicznego innej klasy
        k1.Metoda1();     //OK, wywołanie metody publicznej innej klasy
        k1.y = 22;        //błąd kompilacji, odwołanie do pola prywatnego innej
                        //klasy
        k1.metoda2();     //błąd kompilacji, wywołanie metody prywatnej innej
                        //klasy
        z = 22.1;         //OK, odwołanie do "swojego" pola prywatnego
    }
}
```

W powyższym przykładzie nie przypadkiem nazwy składowych prywatnych rozpoczynają się małą literą, a publiczne wielką. W języku C# przyjęto konwencję, że do nazw składowych prywatnych stosujemy notację CamelCase, natomiast dla nazw składowych publicznych notację PascalCase.

Modyfikator dostępu możemy pominąć. Składowa zostaje wtedy przez domniemanie składową prywatną.

Klasy częściowe

Do wersji języka C# 2.0 definicja klasy była niepodzielna, a co za tym idzie, musiała mieścić się w jednym pliku. W wersji 2.0 języka C# wprowadzono tzw. *klasy częściowe*. Klasę częściową tworzymy przez dodanie przed słowem `class` słowa kluczowego `partial`:

```
//Plik1.cs
partial class Klasa1 {
    //częściowa definicja klasy
}

//Plik2.cs
partial class Klasa1 {
    //częściowa definicja klasy
}
```

Obie części będą tworzyły jedną klasę pod warunkiem, że będą miały tę samą, w pełni kwalifikowaną nazwę, czyli będą znajdować się w tej samej przestrzeni nazw. Wszystkie części muszą być dostępne w czasie kompilacji – nie możemy w ten sposób rozszerzać klas już skompilowanych. Klasy częściowe używane są w środowisku Visual Studio przez generatory kodu (graficzny designer w aplikacjach Windows Forms czy Web Forms). Jeden plik z definicją klasy jest do naszej dyspozycji i możemy dodawać tam własny kod, drugi jest uaktualniany przez środowisko Visual Studio i nie powinniśmy go modyfikować.

W wersji 3.0 języka C# wprowadzono dodatkowo pojęcie metod częściowych. Metody częściowe podlegają następującym regułom:

- muszą być definiowane w klasach częściowych
- typem zwracanym musi być `void`
- nie mogą posiadać modyfikatora dostępu (jak również innych modyfikatorów, takich jak `virtual`, `sealed` czy `new`) – domyślnie prywatne
- mogą być metodami statycznymi lub „obiektu”
- można przekazywać do nich parametry, poza przesyłaniem argumentu jako parametr wyjściowy „out”

W celu demonstracji rozważmy następujący przykład:

```
partial class K1 {  
    partial void f(int x);    //deklaracja metody częściowej  
    public void Metoda() {  
        int a = 10;  
        f(a++);  
        Console.WriteLine(a);  
    }  
}
```

Dopóki nie dostarczymy definicji metody częściowej, linijki zawierające jej wywołanie są pomijane. Wywołanie metody `Metoda` spowoduje wypisanie na ekranie wartości 10, czyli inkrementacja zmiennej `a` nie zostanie wykonana. Gdy dodamy do projektu nowy plik i umieścimy w nim następujący kod:

```
partial class K1{  
    partial void f(int x){    //definicja metody częściowej  
        //ciało metody  
    }  
}
```

po ponownym skompilowaniu projektu, w wyniku wywołania metody `Metoda` na ekranie zostanie wyświetlona wartość 11.

Klasy częściowe mogą być również przydatne w przypadku pracy grupowej. Kilku programistów może pracować nad różnymi aspektami tej samej klasy równolegle.

Struktury

W języku C#, jak to już zostało opisane w kursie „Wprowadzenie do programowania”, nowy typ możemy definiować przy pomocy słowa kluczowego `struct`. Typ ten krótko nazywać będziemy strukturą lub typem strukturalnym. Struktura, w odróżnieniu od klasy, jest typem wartościowym, inaczej typem bezpośrednim (klasa definiuje typ referencyjny). Zmienne typów strukturalnych zawierają w sobie bezpośrednio dane, więc w przypadku struktur ciężko jest mówić o obiekcie, gdyż obiekt jest przechowywany na zarządzanej stercie. Różnice między typem referencyjnym a typem bezpośrednim szczegółowo omówione są w kursie „Wprowadzenie do programowania”. Podsumowując, struktura nie jest klasycznym typem obiektowym, choćby nie można po niej dziedziczyć. Zwróćmy jednak uwagę, że struktura może zawierać pola oraz metody, możemy również sterować dostępem do tych składowych przy pomocy słów kluczowych `public` i `private`. W metodzie zdefiniowanej wewnątrz struktury możemy używać słowa kluczowego `this` (pod warunkiem, że nie jest to metoda statyczna). Możemy też definiować struktury częściowe i definiować wewnątrz nich metody częściowe. Strukturę a nie klasę wybieramy jako implementację typu dla czegoś, co nie będzie zajmować dużo miejsca i nie będzie zarządzane przez automatyczny odśmieczacz pamięci. Struktury bardzo dobrze nadają się do reprezentowania takich pojęć, jak rozmiar czy punkt.

Język UML

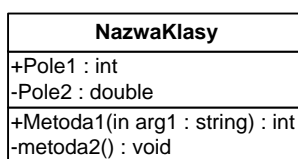
Podobnie jak algorytm możemy rysować przy pomocy schematów blokowych, również klasy możemy projektować w sposób graficzny. Obecnie najbardziej popularnym standardem służącym do opisu problemu w sposób obiektowy jest język UML (ang. *Unified Modeling Language*, czyli Ujednolicony Język Modelowania). W środowisku Visual Studio możemy wygenerować diagram klas (rysunek przedstawiający klasy w naszym projekcie). W tym celu w okienku **Solution Explorer** zaznaczamy element reprezentujący projekt, dla którego chcemy wygenerować diagram klas, naciskamy prawy klawisz myszy i z menu kontekstowego wybieramy **View Class Diagram**. Opcja ta niestety nie jest dostępna w wersji Express. Musimy zwrócić uwagę, że diagram wygenerowany przez Visual Studio nie jest diagramem zgodnym z językiem UML, można powiedzieć, że jest UML-podobny.

W języku UML klasa jest reprezentowana w postaci prostokąta. W prostokącie możemy wyróżnić trzy części: nazwę klasy, pola (atrybuty) oraz metody. Do oznaczenia, że składowa jest publiczna, przed jej nazwą stawiamy znak plus (+), natomiast do oznaczenia składowych prywatnych używamy znaku minus (-).

Następująca definicja klasy:

```
class NazwaKlasy{
    public int Pole1;
    private double pole2;
    public int Metoda1(string arg1){}
    private void metoda2(){}
}
```

w języku UML ma następującą reprezentację:



Rys. 2 Reprezentacja klasy w języku UML

Zwróćmy uwagę, że celem tego kursu nie jest nauka języka UML. UML ułatwia jednak zobrazowanie pewnych zależności między klasami i pokazanie pewnych zagadnień. W kursie będą więc przedstawione tylko niektóre elementy języka UML, bez szczegółowego opisu – te które zostaną użyte.

Przykładowe rozwiązanie

Tworzenie klasy

Naszym zadaniem jest utworzenie klasy *Osoba* posiadającej następujące cechy: imię, nazwisko oraz rok urodzenia.

Klasa ta prawdopodobnie będzie wykorzystywana w programie obsługującym firmę, która produkuje produkty dla osób pełnoletnich. Musimy zapewnić więc, żeby nie można było dodać osoby niepełnoletniej. Klasa musi również zawierać następujące metody:

- **PobierzInformacje** – zwraca dane osoby w postaci łańcucha znaków. Metoda ta powinna zwracać inny napis dla kobiety i mężczyzny. Powinna wykorzystywać prywatną metodę *CzyKobieta*, która określa płeć na podstawie ostatniej litery imienia (w języku polskim imiona żeńskie kończą się literą „a”).
- **UstawRokUrodzenia** – ustawia pole *rokUrodzenia*. W przypadku gdy podany rok urodzenia wskazuje na osobę niepełnoletnią, rzucany jest wyjątek.

Utworzenie klasy Osoba

Poszczególne cechy naszej klasy zaimplementujemy jako odpowiednie pola:

- Imię i Nazwisko jako pole typu string
- rokUrodzenia jako pole typu int

Zauważmy, że pole reprezentujące rok urodzenia musi być polem prywatnym, w celu zapewnienia, że nie zostanie mu nadana niepożądana wartość. Implementacja klasy Osoba może wyglądać następująco:

```
class Osoba {  
    public string Imie;  
    public string Nazwisko;  
    private int rokUrodzenia;  
}
```

Dodanie funkcjonalności do klasy Osoba

Do klasy Osoba dodajmy publiczną metodę UstawRokUrodzenia. Przy pomocy tej metody uzyskamy możliwość nadania wartości prywatnemu polu rokUrodzenia:

```
public class Osoba {  
    ...  
    public void UstawRokUrodzenia(int rokUrodzenia) {  
        if (DateTime.Now.Year - rokUrodzenia < 18)  
            throw new ArgumentException("Osoba musi być pełnoletnia");  
        this.rokUrodzenia = rokUrodzenia;  
    }  
}
```

Do klasy Osoba dodajmy prywatną metodę CzyKobieta. Metodę utworzymy prywatną, ponieważ w programie głównym informacja, czy mamy do czynienia z kobietą, czy z mężczyzną jest nieistotna. Metoda ta jest wykorzystywana tylko przy wypisywaniu informacji o danej osobie.

```
public class Osoba {  
    ...  
    private bool CzyKobieta() {  
        if (Imie.EndsWith("a")) {  
            return true;  
        }  
        return false;  
    }  
}
```

Do klasy Osoba dodajmy publiczną metodę PobierzInformacje:

```
public class Osoba {  
    ...  
    public string PobierzInformacje() {  
        string tytul = "";  
        if (Imie != null){  
            if (CzyKobieta())  
                tytul = "Pani";  
            else  
                tytul = "Pan";  
        }  
        return string.Format("{0} {1} {2} ur. w {3} roku.",  
            tytul, Imie, Nazwisko, rokUrodzenia);  
    }  
}
```


Utworzenie programu testującego

W metodzie Main zaprezentujemy, jak można skorzystać z wcześniej utworzonej klasy Osoba. Sprawdźmy czy możemy skorzystać z prywatnych składowych klasy Osoba. Przykładowy kod metody Main znajduje się poniżej.

```
static void Main(string[] args) {  
    Osoba m = new Osoba();  
    Osoba k = new Osoba();  
    m.Nazwisko = "Kowalski";  
    m.Imie = "Jan";  
    //m.rokUrodzenia = 1990;    //błąd kompilacji  
    k.Imie = "Anna";  
    k.Nazwisko = "Nowak";  
    m.UstawRokUrodzenia(1985);  
    k.UstawRokUrodzenia(1989);  
    Console.WriteLine("k - {0}", k.PobierzInformacje());  
    Console.WriteLine("m - {0}", m.PobierzInformacje());  
    //k.CzyKobieta();    //błąd kompilacji  
    Console.ReadKey();  
    Osoba dziecko = new Osoba();  
    dziecko.UstawRokUrodzenia(DateTime.Now.Year - 16);    //zgłoszenie  
        //wyjątku, osoba niepełnoletnia  
}
```

Definiowanie klas i metod częściowych

Naszym zadaniem jest przetestowanie klas i metod częściowych. Załóżmy, że obiekty klasy Osoba chcemy zapisywać do pliku. Sama metoda zapisująca dane osoby jest zdefiniowana w oddzielnym pliku jako metoda częściowa. Dodatkowo musimy kontrolować, ile razy dany obiekt był zapisywany.

Modyfikacja klasy Osoba

Zaznaczmy, że klasa Osoba jest klasą częściową:

```
partial class Osoba
```

Dodajmy do klasy Osoba pole prywatne `iloscZapisow` typu `int`. Nadajmy temu polu wartość 0. Dodajmy również publiczną metodę zwracającą wartość tego pola:

```
private int iloscZapisow=0;  
public int PobierzLiczbeZapisow() {  
    return iloscZapisow;  
}
```

Dodajmy metodę częściową `zapisz`. Przypomnijmy, że metoda częściowa jest przez domyślność prywatna.

```
partial void zapisz(int licznik);
```

Dodaj metodę publiczną `Zapisz`, która będzie wywoływać metodę częściową `zapisz`:

```
public void Zapisz() {  
    zapisz(++iloscZapisow);  
}
```

W metodzie Main przetestujemy dodany ostatnio kod. Sprawdźmy, czy zmienia się wartość pola `iloscZapisow`. Przykładowy kod metody Main znajduje się poniżej.

```
m.Zapisz();  
m.Zapisz();  
Console.WriteLine("Ilość zapisów do pliku: {0}",m.PobierzLiczbeZapisow());
```

Skompilujmy, uruchommy i przetestujmy działanie programu. Zwróćmy uwagę, że ilość zapisów do pliku wynosi zero.

Dodanie definicji metody częściowej

Dodajmy definicję drugiej części klasy *Osoba*, która będzie zawierać definicję metody częściowej *zapisz*. Definicja ta może znajdować się w oddzielnym pliku. Zwróćmy uwagę, że klasa *Osoba* musi znajdować się w tej samej przestrzeni nazw, co pierwsza część definicji tej klasy. Przykładowa implementacja znajduje się poniżej:

```
partial class Osoba {  
    partial void zapisz(int licznik) {  
        IO.StreamWriter sw = null;  
        string nazwa =  
            string.Format("{0}{1}{2}.txt", Imie, Nazwisko, iloscZapisow);  
        try {  
            sw = new IO.StreamWriter(nazwa);  
            sw.WriteLine("Numer zapisu: {0}", iloscZapisow);  
            sw.WriteLine("Imie: {0}\nNazwisko: {1}", Imie, Nazwisko);  
            sw.WriteLine("Rok urodzenia: {0}", rokUrodzenia);  
        }  
        finally {  
            if (sw != null)  
                sw.Close();  
        }  
    }  
}
```

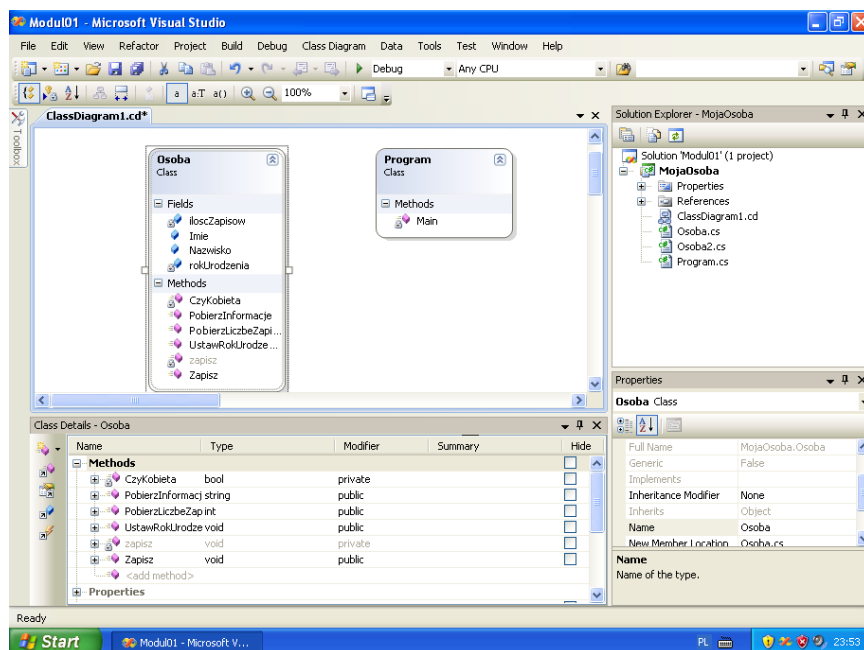
Skompilujmy i przetestujmy działanie programu. Zwróćmy uwagę, że po dodaniu definicji metody częściowej, ilość zapisów do pliku wynosi dwa.

Generowanie i modyfikacja diagramu klas

Naszym zadaniem jest przetestowanie możliwości generacji diagramu klas przy pomocy środowiska Visual Studio oraz jego modyfikacji. Przypomnijmy, że wersja Express nie ma dostępnej tej opcji i jeżeli korzystamy z tej wersji środowiska, przykład nie może być zrealizowany.

Generacja diagramu klas

Zakładając, że poprzedni projekt został utworzony w środowisku Visual Studio, kliknijmy prawym przyciskiem myszy w okienku **Solution Explorer** element reprezentujący projekt. Z menu kontekstowego wybierzmy **View Class Diagram**. Powinniśmy otrzymać diagram podobny jak na rysunku (Rys. 3 Diagram klas wygenerowany w środowisku Visual Studio rys. 3). Zauważmy, że otrzymany diagram różni się od diagramu klas UML, np. do oznaczenia składowej prywatnej jest stosowany rysunek kłódki.



Rys. 3 Diagram klas wygenerowany w środowisku Visual Studio

Modyfikacja diagramu klas

Do diagramu klas dodajmy nową strukturę reprezentującą adres. W tym celu musimy kliknąć prawym klawiszem myszy pusty obszar roboczego diagramu klas i z menu kontekstowego wybrać **Add -> Struct**. W oknie dialogowym **New Struct** w polu **Struct name** wypiszmy nową strukturę **Adres** i naciśnijmy przycisk **OK**. Zauważmy, że środowisko automatycznie utworzyło i dodało do projektu plik **Adres.cs**, w którym znajduje się definicja struktury **Adres**. Spróbujmy teraz dodać do struktury pola reprezentujące: miejscowość, ulicę i numer domu. W tym celu zaznaczmy prostokąt reprezentujący strukturę **Adres**. Następnie w oknie **Class Details – Adres** w gałęzi **Fields** dodajmy publiczne pola:

- Miejscowosc typu string
- Ulica typu string
- NumerDomu typu uint

Zauważmy, że zmiany wprowadzone w diagramie są od razu odzwierciedlane w kodzie.

Przejdźmy do pliku **Osoba.cs** i dodajmy do klasy **Osoba** prywatne pole adres typu **Adres**. Zauważmy, że zmiany wprowadzane w kodzie są również automatycznie odzwierciedlane na diagramie.

Gotowy powyższy projekt znajduje się w katalogu **Demo\Modul01**.

Porady praktyczne

Uwagi ogólne

- Nazwa klasy może składać się z wielkich i małych liter, cyfr, znaku podkreślenia, z tym że nazwa musi być unikalna w danym bloku kodu, nie może rozpoczynać się od cyfry oraz nie może być słowem kluczowym. Pamiętaj, że w języku C# rozróżniamy wielkość liter.
- Tworząc klasę lub strukturę tworzysz nowy typ. Tworząc klasę tworzysz nowy typ referencyjny, tworząc strukturę tworzysz nowy typ bezpośredni.
- Składowa zdefiniowana w dowolnym miejscu klasy lub struktury jest automatycznie dostępna we wszystkich metodach tej klasy lub struktury.
- Definicja klasy musi znajdować się w pojedynczym pliku, chyba że słowo kluczowe **class** poprzedzisz słowem kluczowym **partial**.

- W jednym pliku możesz zdefiniować wiele klas. Zalecane jest jednak, żeby w jednym pliku była zdefiniowana tylko jedna klasa.
- Klasę lub strukturę możesz tworzyć wewnątrz przestrzeni nazw, jak również wewnątrz przestrzeni globalnej oraz wewnątrz innych klas lub struktur. Klasy nie wolno definiować wewnątrz metody.
- W przypadku gdy klasa jest zdefiniowana wewnątrz innej klasy, to metody klasy wewnętrznej mają dostęp do wszystkich składowych klasy zewnętrznej, również do składowych prywatnych.
- Pamiętaj, że jeśli nie zdefiniujesz modyfikatora dostępu do składowej, składowa staje się przez domyślność prywatna. Zalecane jest nawet w przypadku składowych prywatnych określenie modyfikatora dostępu w sposób jawny.
- Obiekty klasy zawsze tworzysz operatorem `new`. Deklaracja zmiennej, której typ jest zdefiniowany przez klasę, nie tworzy automatycznie obiektu tej klasy.
- Utworzenie zmiennej, której typ jest określony przez strukturę, automatycznie powoduje rezerwację pamięci pod jej wszystkie pola.
- Każdy obiekt lub zmienna typu strukturalnego zawiera własną kopię pól. Kod metody występuje w pamięci tylko raz.
- W języku C# przyjęte są pewne konwencje nazewnictwa i powinieneś ich przestrzegać. Dla nazw składowych prywatnych stosujemy notację CamelCase, natomiast dla nazw składowych publicznych – notację PascalCase.
- Metody klasy możesz przeciążać.
- Słowo kluczowe `this` w metodach klasy jest traktowana jako tylko do odczytu. Nie oznacza to jednak, że nie można przy jej pomocy zmieniać wartości pól obiektu, do którego się odnosi.
- W środowisku Visual Studio (również w edycji Express) istnieje specjalne okienko o nazwie **Class View**, które umożliwia w szybki sposób obejrzenie składowych klasy, jak również szybkie przejście do miejsca definicji pola lub metody.

Projektowanie klas

- Tworząc swoje klasy, staraj się unikać sprzężeń między klasami. *Sprzężenie* (ang. *coupling*) jest miarą, jak bardzo obiekty, podsystemy lub systemy zależą od siebie nawzajem. Luźne sprzężenia powoduje, że późniejsze modyfikacje jednej klasy nie wpływają na działanie obiektów drugiej klasy.
- Projektując klasę staraj się, żeby Twoja klasa nie odpowiadała za zbyt dużo zagadnień. Zbyt duża funkcjonalność przydzielona klasie powoduje, że z czasem może się ona nadmiernie rozrosnąć i będzie trudna do zrozumienia, przetestowania oraz późniejszej modyfikacji (pielęgnacji). Miarę, jak funkcjonalnie powiązane są metody danej klasy, nazywamy *spójnością* (ang. *cohesion*). Pamiętaj, aby Twój kod był podzielony na wysoko spójne klasy.
- Do projektowania klas możesz użyć narzędzi graficznych. Istnieje szereg programów, które umożliwiają tworzenie diagramów UML i na ich podstawie generowanie kodu w języku C# (inżynieria wprzód) oraz generowanie diagramów UML z kodu (inżynieria wstecz). Firma Microsoft również dostarcza takie narzędzia, jest to program Visio w połączeniu z Visual Team System. W środowisku Visual Studio od wersji Standard również można generować diagramy UML-podobne. Zmiany na diagramie są od razu odzwierciedlane w kodzie.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- rozumiesz co to jest klasa i co to jest obiekt
- umiesz pokazać zależność między klasą a obiektem
- umiesz wskazać różnicę między klasą a strukturą

- wiesz co to jest metoda i pole
- potrafisz definiować klasy i struktury
- umiesz definiować metody i pola
- wiesz do czego służy słowo kluczowe `this` wewnątrz metody
- rozumiesz na czym polega ukrywanie pól i metod
- wiesz do czego służą słowa kluczowe `public` i `private`
- potrafisz definiować klasy częściowe
- wiesz co to są metody częściowe i potrafisz je definiować

Dodatkowe źródła informacji

1. Daniel Solis, *Illustrated C# 2008*, Apress, 2008

Książka dla tych wszystkich którzy pragną nauczyć się tworzyć programy w języku C#. Zawiera dokładne omówienie tego języka.

2. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka dla programistów chcących nauczyć się programować w języku C#.

3. Jesse Liberty, Brian MacDonald, *C# 2005. Wprowadzenie*, Helion, 2007

Książka, podobnie jak poprzednia, przeznaczona jest dla programistów chcących nauczyć się programować w języku C#. W dużym stopniu pokrywa się z poprzednią pozycją – nie tylko nazwiskiem autora.

4. Stephen C. Perry, *C# i .NET*, Helion, 2006

Książka w porównaniu z poprzednimi kierowana jest do trochę bardziej zaawansowanych programistów. Opisuje C# i .NET Framework w wersji 2.0

5. Andrew Troelsen, *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition*, Apress, 2007

Książka przeznaczona jest dla bardziej zaawansowanych programistów. Czwarte wydanie tej książki opisuje język C# 3.0 i platformę .NET 3.5.

6. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad, jak powinniśmy pisać swoje programy.

7. Codeguru, <http://www.codeguru.pl>

Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.

8. C Sharp Tutorial, http://www.meshplex.org/wiki/C_Sharp_Tutorial

Internetowy kurs języka C#.

9. C# Corner, <http://www.csharpcorner.com>

Portal poświęcony programowaniu w języku C#.

10. Kurs C#, cz. I, http://www.centrumxp.pl/dotNet/20,1,kategoria,Kurs_C_cz_I.aspx

Przystępny kurs języka C# w języku polskim.

Laboratorium podstawowe

Problem 1 (czas realizacji 30 minut)


Twoja firma opracowuje program dla banku. Twoim zadaniem jest stworzenie i przetestowanie klasy Konto. Klasa Konto posiada następujące pola:

- Wlasciciel – typu Osoba (Imie i Nazwisko)
- saldo – typu decimal
- pin – typu int

Pola saldo i pin powinny być zainicjalizowane wartością zero. Pole pin można zmienić tylko podając obecną wartość pola. Wypłatę z konta (zmniejszenie wartości pola saldo) można uzyskać tylko po podaniu prawidłowego PIN-u. Na koncie nie wolno zrobić debetu. Dodaj metodę, która zwraca informacje o koncie, oczywiście pod warunkiem, że został podany prawidłowy PIN.


Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition	<ul style="list-style-type: none">• Otwórz Visual C# 2008 Express Edition.• Z menu wybierz File -> New Project.• Z listy Visual Studio installed templates wybierz Console Application.• W polu Name wpisz Bank.• Kliknij OK.• Z menu wybierz File -> Save Bank.• W polu Location wybierz folder w którym będzie zapisany projekt.• Zaznacz pole wyboru Create directory for solution.• W polu Solution Name wpisz Modul01.• Naciśnij przycisk Save.
2. Dodaj do projektu klasę Konto	<ul style="list-style-type: none">• Z menu wybierz Project -> Add Class.• W oknie dialogowym Add New Item – Bank z listy Visual Studio installed templates wybierz Class.• W polu Name wpisz Konto.• Kliknij OK.
3. Dodaj do projektu klasę Osoba	<ul style="list-style-type: none">• W pliku Konto.cs w przestrzeni nazw Bank tuż przed definicją klasy Konto dodaj definicję klasy Osoba według wymagań: <pre>class Osoba { public string Imie; public string Nazwisko; }</pre>
4. Dodaj implementację klasy Konto	<ul style="list-style-type: none">• Do klasy Konto dodaj pola zgodnie z wymaganiami. Pole saldo i pin powinny być polami prywatnymi: <pre>public Osoba Wlasciciel; private decimal saldo = 0; private int pin = 0;</pre>• Dodaj prywatną metodę, która sprawdza, czy podany PIN jest poprawny: <pre>private bool SprawdzPin(int pin) { if(this.pin == pin) return true; return false; }</pre>• Dodaj metodę, przy pomocy której możesz dokonać wpłat na konto.

	<p>Sprawdź, czy wpłacana kwota nie jest mniejsza od zera, jeżeli tak jest, rzuć wyjątek:</p> <pre>public void DokonajWplaty(decimal kwota) { if(kwota < 0) throw new ArgumentException("Wpłacana kwota nie może być mniejsza od zera"); saldo += kwota; }</pre> <ul style="list-style-type: none"> • Dodaj metodę, przy pomocy której możesz dokonać wypłat z konta. Metoda w zależności od tego, czy podany PIN jest prawidłowy i czy wypłacana kwota nie przekracza salda zwraca napis Operację zakończono pomyślnie albo Operacja anulowana: <pre>public string DokonajWypłaty(decimal kwota,int pin) { if(!SprawdzPin(pin) saldo < kwota) return "Operacja anulowana"; saldo -= kwota; return "Operację zakończono pomyślnie"; }</pre> <ul style="list-style-type: none"> • Dodaj metodę, przy pomocy której możesz zmienić PIN. Metoda niech zwraca wartość logiczną oznaczającą, czy udało się dokonać zmiany: <pre>public bool ZmienPin(int nowy,int stary) { if(SprawdzPin(stary)) { pin = nowy; return true; } return false; }</pre> <ul style="list-style-type: none"> • Dodaj metodę, która zwróci informacje na temat konta. Metoda ta zwróci tę informacje pod warunkiem, że został podany prawidłowy PIN. W przeciwnym wypadku metoda zwróci napis Brak dostępu: <pre>public string PobierzInformacje(int pin) { if(SprawdzPin(pin)) return string.Format("Pan(i) {0} {1} posiada na koncie: {2}", Wlasciciel.Imie, Wlasciciel.Nazwisko, saldo); return "Brak dostępu"; }</pre>
<p>5. Przetestuj klasę Konto</p>	<ul style="list-style-type: none"> • Przejdź do pliku Program.cs. • Do metody Main dodaj kod, który utworzy tablicę obiektów typu Konto, a następnie przetestuje, czy klasa konto działa prawidłowo. Przykładowy kod jest zamieszczony poniżej: <pre>Konto[] konta = new Konto[2]; konta[0] = new Konto(); konta[1] = new Konto();konta[0].Wlasciciel = new Osoba(); konta[1].Wlasciciel = new Osoba(); konta[0].Wlasciciel.Imie = "Jan"; konta[0].Wlasciciel.Nazwisko = "Kowalski"; konta[1].Wlasciciel.Imie = "Ala"; konta[1].Wlasciciel.Nazwisko = "Kot"; Console.WriteLine("Próba zmiany pinu: {0}", konta[0].ZmienPin(1111, 0)); Console.WriteLine("Próba zmiany pinu: {0}", konta[1].ZmienPin(1111, 1111)); Console.WriteLine("Dokonujemy wpłat:"); konta[0].DokonajWplaty(1200); konta[1].DokonajWplaty(2200); Console.WriteLine("Dokonujemy wypłaty: {0}",</pre>



	<pre> konta[0].DokonajWypłaty(300,1111)); Console.WriteLine("Dokonyjemy wypłaty: {0}", konta[0].DokonajWypłaty(3000, 1111)); Console.WriteLine("Dokonyjemy wypłaty: {0}", konta[1].DokonajWypłaty(200, 1111)); Console.WriteLine("Informacje o koncie: {0}", konta[0].PobierzInformacje(1111)); Console.WriteLine("Informacje o koncie: {0}", konta[1].PobierzInformacje(1111)); Console.WriteLine("Informacje o koncie: {0}", konta[1].PobierzInformacje(0)); Console.ReadKey(); </pre>
6. Skompiluj, a następnie uruchom program	<ul style="list-style-type: none"> • Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. • W celu uruchomienia programu, z menu Debug wybierz Start Debugging.
7. Sprawdź, czy możesz uzyskać dostęp do składowych prywatnych klasy	<ul style="list-style-type: none"> • Na końcu metody Main dodaj następujący kod: <pre> konta[0].saldo = 2000; konta[0].pin = 2345; konta[0].SprawdzPin(1111); </pre> • Spróbuj skompilować projekt. •  Jaki błąd zgłaszany jest przez kompilator? • Skasuj lub umieść w komentarzu dodane ostatnio trzy linijki.


Problem 2 (czas realizacji 15 minut)

Masz napisaną klasę **Kolejka**. Klasa ta implementuje kolejkę FIFO. Twoim zadaniem jest przerobienie jej tak, aby była w pełni obiektowa. Musisz ukryć pola klasy oraz metody pozbawić modyfikatora **static**. Algorytm zastosowany do implementacji kolejki FIFO za pomocą tablicy jest omówiony w kursie „Wprowadzenie do programowania”.

Zadanie	Tok postępowania
1. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. • W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. • W polu Name wpisz TestKolejki. • Kliknij OK.
2. Zaznacz projekt TestKolejki jako projekt startowy	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestKolejki, a następnie z menu kontekstowego wybierz Set as StartUp Project.
3. Do projektu TestKolejki dodaj plik gdzie zaimplementować a jest kolejka FIFO	<ul style="list-style-type: none"> • Z menu wybierz Project -> Add Existing Item. •  Zwróć uwagę, aby przed wykonaniem powyższego polecenia w okienku Solution Explorer był zaznaczony projekt TestKolejki. • W oknie dialogowym Add Existing Item – TestKolejki wybierz plik KolejkaFIFO.cs, który znajduje się w katalogu Kurs\Lab\Start\Modul01, gdzie Kurs jest katalogiem, w którym zainstalowano pliki kursu. • Kliknij Add.
4. Zmień	<ul style="list-style-type: none"> • Przejdź do pliku KolejkaFIFO.cs.

<p>implementację klasy Queue na obiektową</p>	<ul style="list-style-type: none">• Zmień wszystkie pola składowe na pola klasy Kolejka prywatne. W wyniku tej operacji powinieneś otrzymać następujący kod:<pre>private string[] elementy; private int pierwszy; private int ostatni;</pre>• Usuń z definicji metody Utworz słowo static oraz pierwszy argument metody, a następnie zmodyfikuj odpowiednio kod tej metody. Kod metody Utworz po modyfikacjach powinien wyglądać następująco:<pre>public void Utworz(int liczbaElementow){ pierwszy = ostatni = -1; elementy = new string[liczbaElementow]; }</pre>• Usuń z definicji metody CzyPelna słowo static oraz argument metody, a następnie zmodyfikuj odpowiednio kod tej metody. Kod metody CzyPelna po modyfikacjach powinien wyglądać następująco:<pre>public bool CzyPelna() { return ((pierwszy == 0 && ostatni == elementy.Length - 1) pierwszy == ostatni + 1); }</pre>• Usuń z definicji metody DodajDoKolejki słowo static oraz pierwszy argument metody, a następnie zmodyfikuj odpowiednio kod tej metody. Kod metody DodajDoKolejki po modyfikacjach powinien wyglądać następująco:<pre>public void DodajDoKolejki(string element) { if (CzyPelna()) throw new InvalidOperationException("Kolejka jest pełna"); if (ostatni == elementy.Length - 1 ostatni == -1) { elementy[0] = element; ostatni = 0; if (pierwszy == -1) pierwszy = 0; } else elementy[++ostatni] = element; }</pre>• W analogiczny sposób zmień definicję pozostałych metod klasy Kolejka. Kod po modyfikacjach powinien wyglądać następująco:<pre>public bool CzyPusta() { return pierwszy == -1; } public string UsunZKolejki() { if (CzyPusta()) throw new InvalidOperationException("Kolejka jest pusta"); string tmp; tmp = elementy[pierwszy]; if (pierwszy == ostatni) ostatni = pierwszy = -1; else if (pierwszy == elementy.Length - 1) pierwszy = 0; else pierwszy++; return tmp; } public void Wyczysc() {</pre>
---	---

	<pre> Utworz(elementy.Length); } public string SprawdzElement() { if (CzyPusta()) throw new InvalidOperationException("Kolejka jest pusta"); return elementy[pierwszy]; } public int PobierzLiczbeElementow() { if(pierwszy == -1) return 0; if (pierwszy <= ostatni) return ostatni - pierwszy + 1; return elementy.Length - pierwszy + ostatni + 1; } </pre> <p> Którą wersję kodu – przed czy po modyfikacji – uważasz za prostszą (łatwiejszą w pisaniu, analizie)? Która jest według Ciebie bezpieczniejsza? Dlaczego?</p>
<p>5. Popraw program główny tak, aby korzystał ze zmodyfikowanej wersji klasy Kolejka</p>	<ul style="list-style-type: none"> • Usuń z projektu plik Program.cs <ul style="list-style-type: none"> – w okienku Solution Explorer zaznacz plik Program.cs – naciśnij klawisz Delete – w okienku dialogowym naciśnij przycisk OK • Z menu wybierz Project -> Add Existing Item. <p> Zwróć uwagę, aby przed wykonaniem powyższego polecenia w okienku Solution Explorer był zaznaczony projekt TestKolejki.</p> • W oknie dialogowym Add Existing Item – TestKolejki wybierz plik Program.cs, który znajduje się w katalogu Kurs\Lab\Start\Modul01, gdzie Kurs jest katalogiem, w którym zainstalowano pliki kursu. • Kliknij Add. • Przejdź do pliku Program.cs. • Zmodyfikuj kod metody Main tak, aby korzystał ze zmodyfikowanej wersji klasy Kolejka: <pre> static void Main(string[] args) { Kolejka mojaKolejka = new Kolejka(); mojaKolejka.Utworz(10); string tmp; char c; do { c = Menu(); switch (c){ case 'a': case 'A': Console.Write("Podaj napis który ma być dodany do kolejki: "); tmp = Console.ReadLine(); mojaKolejka.DodajDoKolejki(tmp); break; case 'b': case 'B': Console.WriteLine("Napis wyjęty z kolejki: {0}", mojaKolejka.UsunZKolejki()); Console.ReadKey(); break; case 'c': case 'C': </pre>

	<pre> Console.WriteLine("Liczba elementów w kolejce wynosi: {0}", mojaKolejka.PobierzLiczbeElementow()); Console.ReadKey(); break; case 'd': case 'D': mojaKolejka.Wyczysc(); Console.WriteLine("Kolejka wyczyszczona!!!"); Console.ReadKey(); break; } } while (!(c == 'k' c == 'K')); }</pre> <p> Który sposób korzystania z klasy Kolejka uważasz za prostszy?</p>
6. Skompiluj i uruchom program	<ul style="list-style-type: none">• Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program.• W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 90 min)

Twój kolega ma spory zbiór płyt muzycznych. Chce je uporządkować. Poprosił Ciebie, abyś napisał mu program, który pozwoli mu zarządzać tym zbiorem, czyli taką bazę płyt. O każdej płycie chce posiadać następujące informacje:

- tytuł płyty
- typ płyty (czy płyta DVD czy CD)
- czas trwania płyty
- spis utworów znajdujących się na każdej płycie
- spis wykonawców, którzy wykonują utwory na danej płycie
- numer płyty (identyfikator płyty nadany przez kolegę)

O każdym utworze kolega chce wiedzieć:

- tytuł utworu
- czas trwania utworu
- spis wykonawców, którzy wykonują dany utwór na danej płycie
- kompozytora danego utworu
- numer utworu na płycie

Program powinien umożliwić:

- dodanie płyty do bazy danych
- wyświetlenie wszystkich płyt znajdujących się w bazie danych (same tytuły płyt)
- wyświetlenie szczegółowej informacji na temat płyty (tytuł płyty, typ płyty, czas trwania płyty, spis tytułów utworów znajdujących się na płycie z podanym numerem utworu na płycie)
- wyświetlenie wykonawców, którzy wykonują utwory na danej płycie
- wyświetlenie szczegółowej informacji na temat wybranego utworu z danej płyty (tytuł utworu, czas trwania utworu, spis wykonawców którzy wykonują dany utwór, kompozytora danego utworu)
- zapisać bazę do pliku
- odczytać bazę z pliku

Twoim pierwszym zadaniem jest opracowanie diagramu klas. Przed przystąpieniem do implementacji musisz uzyskać akceptację kolegi (prowadzącego), czy projekt jest zgodny z jego oczekiwaniami.

Po uwzględnieniu uwag kolegi (prowadzącego) zaimplementuj program.

Zadanie 2 (czas realizacji 30 min)

Masz napisać program który zawiera bazę danych osób. W celu poprawienia wydajności wyszukiwania osób na strukturę danych, w której będą przechowywane osoby, wybrano drzewo binarne. Masz już szkielet aplikacji. Niestety nie jest napisana obiektowo. Przed dalszą implementacją bazy osób musisz przerobić program tak, aby był obiektowy. Pliki startowe do tego zadania znajdują się w katalogu **Kurs\Lab\Start\Modul01**, gdzie **Kurs** jest katalogiem, w którym zainstalowano pliki kursu. Są to:

- **Biblioteka.cs** – znajduje się tu implementacja klasy osoba
- **Drzewa.cs** – implementacja drzewa przechowującego osoby
- **TestDrzewa.cs** – program testujący drzewo

Drzewo binarne jest omówiony w kursie „Wprowadzenie do programowania”.

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 2

Wersja 2

Konstruktor

Spis treści

Konstruktor	1
Informacje o module	2
Przygotowanie teoretyczne	3
Przykładowy problem	3
Podstawy teoretyczne	3
Przykładowe rozwiązanie	9
Porady praktyczne	13
Uwagi dla studenta	14
Dodatkowe źródła informacji	14
Laboratorium podstawowe	16
Problem 1 (czas realizacji 40 min)	16
Problem 2 (czas realizacji 30 min)	20
Laboratorium rozszerzone	23
Zadanie 1 (czas realizacji 90 min)	23

Informacje o module

Opis modułu

W tym module zapoznasz się z pojęciem konstruktora oraz nauczysz się, jak definiować go w języku C#. Omówiona zostanie również lista inicjalizacyjna konstruktora. Poznasz, co to są inicjalizatory obiektów i jak ich używać. Dowiesz się również co to jest destruktor. Zostanie również wprowadzone pojęcie wzorców projektowych i przedstawiony wzorec projektowy o nazwie *prototyp*.

Cel modułu

Celem modułu jest pokazanie, jak w języku C# można inicjalizować obiekty oraz wprowadzenie pojęcia wzorców projektowych.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- znał pojęcie konstruktora i potrafił go definiować
- znał pojęcie destruktora
- wiedział, co to jest lista inicjalizacyjna konstruktora
- umiał korzystać z inicjalizatorów obiektów
- rozumiał pojęcie wzorców projektowych
- wiedział, co to jest wzorec projektowy prototyp

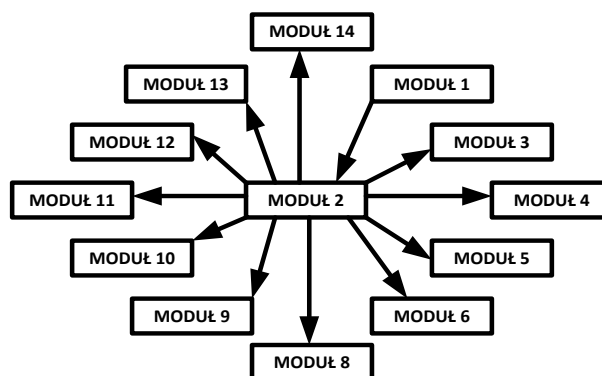
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinienś:

- wiedzieć, co to klasa i co to jest obiekt
- potrafić definiować klasę
- potrafić tworzyć obiekty klasy
- wiedzieć, co to jest pole i co to jest metoda
- znać modyfikatory dostępu public i private
- umieć korzystać z modyfikatorów dostępu

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w module „Pojęcie klasy”.



Rys. 4 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Wiesz, że można kontrolować dostęp do składowych klasy i w ten sposób nie pozwolić wprowadzić obiektu w niedozwolony stan. Wszystko fajnie, ale jak zapewnić, aby obiekt tuż po utworzeniu miał prawidłowy stan? Dostarczenie metody typu *inicjalizuj* czy *utwórz* nie rozwiązuje tego problemu do końca. Zapewne nie chcesz, aby od dobrej woli programisty – użytkownika Twojej klasy – zależało, czy wywoła on odpowiednią funkcję, nadającą prawidłowe wartości poszczególnym polom klasy, czy też zacznie używać obiektu, który nie został prawidłowo zainicjalizowany. Scedowanie odpowiedzialności na użytkownika klasy może prowadzić do nieprzewidywanych skutków. Człowiek jeżeli ma popełnić błąd, to prawdopodobnie go popełni, a dobrze zaprojektowana rzecz, nie tylko w świecie informatyki, zapobiega przypadkowym uszkodzeniom podczas normalnej obsługi. Marzy Ci się pewnie metoda, która byłaby wywoływana automatycznie podczas tworzenia obiektu. Metoda, która jest wywoływana zawsze podczas tworzenia obiektu, której nie można pominąć czy wyłączyć. W językach programowania obiektowego istnieje metoda, która spełnia Twoje marzenie. Jest to konstruktor. Co więcej, możemy również zdefiniować metodę, która jest wywoływana przed zwolnieniem pamięci przez obiekt. Jest to destruktor. Jeśli chcesz wiedzieć więcej, zapoznaj się z treścią tego modułu.

Być może nurtuje Cię również pytanie, czy w programowaniu obiektowym istnieją gotowe przepisy, schematy rozwiązujące określone problemy. W programowaniu strukturalnym mamy do dyspozycji szereg algorytmów oraz różnorodne struktury, programując obiektowo możemy natomiast sięgnąć po wzorce projektowe. W tym module temat ten zostanie tylko wprowadzony, będzie się on jednak często przewijał w pozostałych modułach tego kursu.

Podstawy teoretyczne

W językach programowania obiektowego istnieje specyficzna metoda, która wywoływana jest automatycznie podczas tworzenia obiektu w celu inicjalizacji jego pól. Nazywamy ją *konstruktorem*. W języku C# konstruktor to metoda, która posiada taką samą nazwę, jak nazwa klasy, w której jest zdefiniowany. Jego drugą charakterystyczną cechą jest to, że w jego sygnaturze nie wolno podawać typu wartości zwracanej – zabronione jest nawet użycie słowa kluczowego `void`. Przykład klasy, która zawiera definicję konstruktora, jest umieszczony poniżej:

```
class NazwaKlasy {  
    ...  
    public NazwaKlasy() {                //konstruktor bezparametrowy  
        //ciało  
    }  
    public NazwaKlasy(int arg1,double arg2) {    //konstruktor z parametrami  
        //ciało  
    }  
}
```

Jak widać w powyższym kodzie, klasa może mieć kilka konstruktorów – czyli konstruktor, podobnie jak zwykłą metodę, możemy przeciążyć. Możemy dostarczyć w ten sposób różne sposoby inicjalizacji obiektów danej klasy.

Konstruktor, jak już wspomniano, jest wywoływany tuż po utworzeniu obiektu. Do wywołania konstruktora używamy operatora `new`, nie możemy natomiast wywołać go na rzecz obiektu:

```
NazwaKlasy x1 = new NazwaKlasy(); //wywołanie konstruktora bezparametrowego  
NazwaKlasy x1 = new NazwaKlasy(2, 3.3); //wyw. konstruktora z parametrami
```

Powyższy kod może niektórych wprawić w zdziwienie: „Przecież operatora `new` używałem niejednokrotnie, a nie definiowałem żadnego konstruktora”. Zgadza się. Konstruktor bezparametrowy jest specyficzną metodą, która jest automatycznie generowana przez kompilator.

Konstruktor automatycznie wygenerowany nazywany jest *konstruktorem domyślnym*. W przypadku klas istnieje jednak sytuacja, gdy nie zostanie on wygenerowany. Możemy zablokować automatyczne wygenerowanie przez kompilator konstruktora domyślnego przez zdefiniowanie dowolnego konstruktora dla danej klasy, co ilustruje poniższy przykład:

```
class NazwaKlasy {  
    ...  
    public NazwaKlasy(int arg1,double arg2) {    //konstruktor z parametrami  
                                                //ciało  
    }  
}  
...  
static void Main() {  
    NazwaKlasy x = new NazwaKlasy();    //błąd kompilacji, brak konstruktora  
                                        //domyślnego  
}
```

W sytuacji, gdy któremuś polu nie przypiszemy wartości w sposób jawny w konstruktorze klasy, zostanie mu nadana automatycznie wartość domyślna odpowiednia dla typu danego pola. Wartość domyślną będą również miały pola, w przypadku skorzystania z konstruktora domyślnego. Wartością domyślną dla typów liczbowych jest 0, dla typu logicznego `false`, zaś dla typów referencyjnych `null`, a w przypadku zmiennych składowych, których typ określony jest przez strukturę, pola tej struktury inicjalizowane są wartością reprezentującą zero dla danego typu.

W powyższych przykładach wszędzie definiowaliśmy konstruktor jako publiczny. Konstruktor możesz zdefiniować z innym modyfikatorem. Przykład zastosowania prywatnego konstruktora możemy znaleźć w module 4 tego kursu.

Wiemy już, że klasa może mieć kilka konstruktorów. Różnią się one zazwyczaj w nieznacznym sposób. Język C# daje nam możliwość wywołania jednego konstruktora przez drugi. Służy do tego *lista inicjalizacyjna* konstruktora. Lista inicjalizacyjną tworzymy przez umieszczenie dwukropka po nawiasie zamykającym listę parametrów. W języku C# na liście inicjalizacyjnej możemy umieścić wywołanie innego konstruktora danej klasy lub konstruktora klasy bazowej. Pojęcie klasy bazowej zostanie omówione w module 6 tego kursu. Do wywołania innego konstruktora danej klasy używamy słowa kluczowego `this`:

```
class NazwaKlasy {  
    private Typ1 pole1;  
    private Typ2 pole2;  
    private Typ3 pole3  
    public NazwaKlasy(): this(war1)  
    { }  
    public NazwaKlasy(Typ1 arg1):this(arg1, war2)  
    { }  
    public NazwaKlasy(Typ1 arg1, Typ2 arg2):this(arg1, arg2, war3)  
    { }  
    public NazwaKlasy(Typ1 arg1, Typ2 arg2, Typ3 arg3) {  
        pole1 = arg1;  
        pole2 = arg2;  
        pole = arg3;  
    }  
}
```

Warto zwrócić uwagę na pewne ograniczenia. Konstruktor nie może wywoływać sam siebie i w wywołaniu konstruktorów nie może powstać pętla. Podsumowując, w wywołaniu konstruktora przez konstruktor nie może wystąpić rekurencja.

Polu klasy możemy również nadać wartość w miejscu jego definicji. Wartość ta nadpisuje wartość automatycznie nadaną temu polu (wartość domyślną dla danego typu). Dotyczy to również

automatycznie generowanego konstruktora. Wartość ta jest jednak nadpisywana, jeżeli w konstruktorze przypiszemy odpowiedniemu polu inną wartość. Pokażmy to na przykładzie:

```
class K {
    private int x = 10;
    private int y = 20;
    private int z;

    public K() {
        x = 100;
    }

    static void Main() {
        K k = new K();
        Console.WriteLine(k.x);    //zostanie wypisane 100, wartość nadana
                                   //przez konstruktor
        Console.WriteLine(k.y);    //zostanie wypisane 20, wartość nadana w
                                   //miejscu definicji pola
        Console.WriteLine(k.z);    // zostanie wypisane 0, wartość nadana
                                   // automatycznie przez kompilator
    }
}
```

W wersji języka C# 3.0 została wprowadzona nowa konstrukcja, *inicjalizatory obiektów* (ang. *Object Initializers*), która umożliwia nadanie wartości publicznym polom i właściwościom obiektu w miejscu jego utworzenia, w sposób podobny do tego, jak inicjalizowane są tablice. Właściwości klasy zostaną omówione w module 3 tego kursu. Inicjalizatory obiektów możesz również stosować w odniesieniu do zmiennych, których typ zdefiniowany jest przez strukturę. Konstrukcję tę najlepiej pokazać na przykładzie:

```
class K1 {
    public int A;
    public int B;
}

class K2 {
    public int X;
    public int Y;
    public K1 K;

    public K2() {
        X = 100;
        Y = 200;
        K = new K1();
        K.A = 10;
        K.B = 20;
    }

    public K2(int x, int y, K1 k) {
        X = x;
        Y = y;
        K = k;
    }

    static void Main() {
        K2 x = new K2 { X = 20, K = new K1 {A = 20, B = 30 } };
        K1 k1 = new K1() {A = 20, B = 30 };
        K2 y = new K2(1, 2, k1) {Y = 10, X = 50 };
    }
}
```

Analizując powyższy kod możemy zauważyć, że:

- nie wszystkie pola muszą być inicjalizowane w inicjalizatorze
- inicjalizatory mogą być zagnieżdżone
- nie jest ważna kolejność pól w inicjalizatorze

Powyższy kod metody Main możemy wyobrazić sobie w następujący sposób:

```
static void Main() {  
    K2 x = new K2();  
    x.X = 20;  
    x.K = new K1();  
    x.K.A = 20;  
    x.K.B = 30;  
    K1 k1 = new K1();  
    k1.A = 20;  
    k1.B = 30;  
    K2 y = new K2(1, 2, k1);  
    y.Y = 10;  
    y.X = 50;  
}
```

Inicjalizatory obiektów są tylko słodzikiem, ale słodzikiem bardzo wygodnym. Docenimy je na pewno, zwłaszcza gdy będziemy tworzyli obiekty klas, dla których ich twórca nie dostarczył odpowiedniego konstruktora.

Inicjalizacja struktur

Dla struktur możesz również definiować konstruktory, są jednak pewne różnice. Konstruktor domyślny (bezparametrowy) jest zawsze generowany.

```
struct NazwaStruktury {  
    ...  
    public NazwaStruktury(int arg1, double arg2) {  
        //konstruktor z parametrami  
    }  
}  
...  
static void Main() {  
    NazwaStruktury x = new NazwaStruktury ();  
}
```

Powyższy kod nie spowoduje błędu kompilacji, jak to by było w przypadku klas.

Dla struktur nie możemy jednak dostarczyć własnej wersji konstruktora bezparametrowego. Poniższy kod spowoduje błąd kompilacji:

```
struct NazwaStruktury {  
    public NazwaStruktury() {    //błąd, nie wolno definiować konstruktora  
        //bezparametrowego dla struktur  
    }  
}
```

Konstruktor domyślny struktury inicjalizuje wszystkie pola wartością domyślną dla danego typu.

Następną różnicą jest, że jeżeli zdecydujemy się zdefiniować własny konstruktor, to musimy w nim nadać wartości wszystkim polom danej struktury. W przypadku struktur nie ma automatycznej inicjalizacji pól wartością domyślną:

```
struct NazwaStruktury {  
    private int pole1;  
    private int pole2;  
    public NazwaStruktury(int arg1) {
```

```
        pole1 = arg1;
    }
}
```

Powyższy kod spowoduje błąd kompilacji. Pole `pole2` nie ma nadanej wartości w konstruktorze. Błąd ten w najprostszy sposób poprawić, przez wywołanie konstruktora domyślnego:

```
public NazwaStruktury(int arg1): this()
```

W strukturach nie wolno również inicjalizować pola w miejscu jego definicji:

```
struct NazwaStruktury {
    public int pole1 = 20;    //błąd kompilacji!
}
```

Finalizator

W programowaniu obiektowym zostało wprowadzona metoda, która działa niejako na drugim końcu życia obiektu. Jest ona wywoływana tuż przed zwolnieniem pamięci przydzielonej dla obiektu w celu „posprzątania” po nim. Metodę tą nazywamy *destruktor*. W języku C# możemy zdefiniować metodę, która pełni podobną rolę. Metodę tą nazywamy *finalizatorem*. Finalizator posiada nazwę taką samą jak nazwa klasy, w której jest zdefiniowany poprzedzoną znakiem tyldy (~). Podobnie jak dla konstruktora, dla finalizatora nie określamy wartości zwracanej. Dodatkowo finalizator nie może mieć modyfikatora dostępu oraz nie można do niego przesłać argumentów, a co za tym idzie, nie można go przeciążyć. Klasa może mieć tylko jeden finalizator:

```
class NazwaKlasy {
    ...
    ~NazwaKlasy() {    //definicja finalizatora
        ...
    }
}
```

Finalizator jest wywoływany przez proces automatycznego zwalniania pamięci, tzw. *Garbage Collector*. Zdefiniowanie tej metody dla klasy powoduje, że zwalnianie pamięci zajmowanej przez obiekt potrzebuje co najmniej dwóch uruchomień procesu automatycznego zwalniania pamięci. Wpływa to niewątpliwie na wydajność. Powinniśmy się w związku z tym dobrze zastanowić, zanim zdefiniujemy finalizator. W innych językach programowania niż język C# destruktor często definiowany jest w celu zwolnienia pamięci przydzielonej dynamicznie w konstruktorze. W języku C# zwalnianiem pamięci przydzielonej dynamicznie zajmuje się Garbage Collector, więc w tych sytuacjach definicja finalizatora jest niepotrzebna. Drugą cechą, na którą należy zwrócić uwagę przy definiowaniu finalizatora jest to, że nie jesteśmy w stanie określić, kiedy zostanie on wywołany, ponieważ nie wiemy kiedy zostanie uruchomiony proces automatycznego zwalniania pamięci. Mówimy o tym krótko, że w języku C# finalizator jest metodą niedeterministyczną – niewiadomo kiedy będzie uruchomiony. Niedeterminizm finalizatora jest prawdopodobnie jedną z przyczyn wprowadzenia nowego pojęcia w języku C#, zastępującego słowo destruktor. W pewien sposób możemy jednak sterować procesem automatycznego zwalniania pamięci przy pomocy klasy GC i jej metod. W celu uruchomienia Garbage Collectora możemy wywołać jej metodę `Collect`. Mimo to nadal nie możemy być pewni, w jakiej kolejności zostaną wywołane finalizatory. W celu uzyskania determinizmu przy pracach porządkowych przed zwolnieniem pamięci zajmowanej przez obiekt, w .NET Framework używa się interfejsu `IDisposable` i jego metody `Dispose`. Zazwyczaj finalizator implementujemy razem z interfejsem `IDisposable`, dlatego przykład implementacji finalizatora jest pokazany dopiero w module 11 tego kursu.

Warto zapamiętać, że w języku C# destruktora nie możemy definiować dla struktur.

Wzorce projektowe

Tworzenie oprogramowania metodą zorientowaną obiektowo, szczególnie projektowanie, nie jest łatwym zadaniem. Pewne problemy często się powtarzają. Są problemami typowymi. Dobrze by było nie wywarzać już otwartych drzwi. Niejako z tej idei zrodził się pomysł *wzorców projektowych* (ang. *design patterns*). Wzorce projektowe definiujemy jako zbiór sprawdzonych w praktyce uniwersalnych rozwiązań często pojawiających się rzeczywistych problemów projektowych. W informatyce wzorce projektowe zyskały bardzo na popularności dzięki książce „Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku”, którą napisali Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides, nazywani często „Bandą Czworka” (ang. *Gang of Four*). Autorzy dla każdego wzorca projektowego definiują cztery elementy, które go charakteryzują. Są to:

- **Nazwa wzorca** – nazwa jednoznacznie identyfikująca wzorzec, składa się z jednego lub dwóch słów. Wprowadza nowe pojęcie do słownika, za pomocą którego możemy rozmawiać o projekcie. Nazwy wzorców tworzą nową warstwę abstrakcji, przy pomocy której możemy opisywać swój projekt na wyższym poziomie abstrakcji.
- **Problem** – opisuje, kiedy stosować dany wzorzec z całym kontekstem.
- **Rozwiązanie** – opisuje, jak rozwiązać dany problem projektowy. Nie jest to jednak opis konkretnej implementacji. Zamiast tego podane są klasy, które powinny być uwzględnione wraz z ich odpowiedzialnością (do czego służą) oraz opisem zależności między poszczególnymi klasami i opisem zasad współpracy między obiektami tych klas. Często w celu zobrazowania rozwiązania stosuje się diagram UML.
- **Skutki** – konsekwencje dokonania wyboru danego wzorca projektowego. Zawierają one między innymi opis wpływu zastosowania danego wzorca na elastyczność systemu, rozszerzalność czy przenośność.

W książce autorzy podzielili również wzorce na następujące trzy kategorie:

- **Wzorce konstrukcyjne** – skupiają się na procesie tworzenia obiektu.
- **Wzorce strukturalne** – skupiają się budowie oraz organizacji klas i obiektów. Pomagają łączyć obiekty w większe struktury.
- **Wzorce czynnościowe** – skupiają się na opisie interakcji obiektów i klas oraz podziale odpowiedzialności między klasami.

Teraz zapoznamy się z pierwszym wzorcem projektowym o nazwie *prototyp*. Prototyp jest wzorcem konstrukcyjnym. Służy do utworzenia nowego obiektu, ale nie od podstaw, ale w oparciu o inny obiekt wzorcowy. Ten inny obiekt wzorcowy stanowi prototyp nowo tworzonego obiektu, stąd nazwa wzorca. Tworzenie wzorca w oparciu o prototyp może być szczególnie przydatne, gdy utworzenie obiektu wymaga czasochłonnych czynności np. połączenia z bazą danych czy wykonania skomplikowanych obliczeń. Nie omówiliśmy jeszcze pojęcia dziedziczenia oraz interfejsu, dlatego zostanie tu przedstawiona pewna uproszczona wersja implementacji wzorca projektowego prototyp. Głównym sposobem implementacji tego wzorca jest dodanie metody do klasy prototypu, która zwraca żądany obiekt. W przypadku gdy klasa prototypu jest taka sama jak klasa nowego obiektu, wtedy tą metodę nazywa się `Clone` (klonuj), czyli klonowanie polega na utworzenie kopii obiektu w oparciu o obiekt istniejący. Wyróżniamy dwa rodzaje klonowania:

- **Kopiowanie płytkie** (ang. *shallow copy*) – pola typów wartości kopiowane są bit po bicie. W przypadku gdy pole obiektu jest typu referencyjnego, to dane pole w prototypie i odpowiednie pole w nowo utworzonym obiekcie są dowiązane do tego samego obiektu.
- **Kopiowanie głębokie** (ang. *deep copy*) - w przypadku gdy pole obiektu jest typu referencyjnego, to dane pole w prototypie wskazuje na inny obiekt niż odpowiednie pole w nowo utworzonym obiekcie. Pola typów wartości kopiowane są bit po bicie.

W języku C# istnieje metoda, która wykonuje płytkie kopiowanie. Jest to metoda `MemberwiseClone`. Tą metodę posiadają wszystkie typy w języku C#, jest ona odziedziczona po typie `object`. Szczegóły na temat dziedziczenia zostaną omówione w module 6 tego kursu. Możemy przyjąć, że dla każdego typu ta metoda jest prywatna, czyli można ją wywołać tylko z metod danej klasy (tak naprawdę metoda jest `protected`, szczegóły również w module 6). Przykład metody wykonującej płytkie kopiowanie przy użyciu metody `MemberwiseClone` jest umieszczony poniżej:

```
class NazwaKlasy {  
    ...  
    public NazwaKlasy Clone() {  
        return (NazwaKlasy) this.MemberwiseClone();  
    }  
}
```

W powyższym kodzie konieczne jest rzutowanie, ponieważ metoda `MemberwiseClone` przekazuje wartość typu `object`. Szczegóły w module 9 tego kursu.

Do kopiowania głębokiego można wykorzystać mechanizm *serializacji*, który zostanie opisany w module 14 tego kursu. Możemy też stworzyć własny algorytm głębokiego kopiowania obiektu, co zostanie przedstawione w poniższych przykładach oraz w laboratorium.

W języku C# został stworzony specjalny interfejs `ICloneable`, posiadający metodę `Clone`, który służy do tworzenia kopii obiektów. O predefiniowanych interfejsach w bibliotece .NET można znaleźć więcej w module 11.

Kolejnym sposobem na utworzenie obiektu na podstawie innej instancji tej samej klasy jest zdefiniowanie konstruktora, który przyjmuje jako parametr zmienną takiego samego typu, co klasa, w której zdefiniowany jest konstruktor.

Uwaga dla programistów C++: w języku C# konstruktor jest to metoda, która jest wywoływana tylko przy pomocy operatora `new`. Przy przesyłaniu argumentów przez wartość nie jest wywoływany konstruktor. W przypadku przesyłania typów referencyjnych są kopiowane referencje, w przypadku przesyłania typów wartości następuje kopiowanie bit po bicie.

Przykładowe rozwiązanie

Załóżmy, że naszym zadaniem jest utworzenie klasy `Czołg`. Klasa `Czołg` zawiera takie parametry jak:

- numer czołgu typu `int`
- nazwa czołgu typu `string`
- działo typu `Działo` – typ `Działo` jest klasą, która zawiera pole kaliber typu `double`
- pozycja typu `Punkt` – typ `Punkt` jest strukturą, która zawiera współrzędną `x` oraz `y` typu `int`

Tworzenie i wykorzystanie konstruktorów

W pierwszym etapie mamy utworzyć tylko konstruktory, które będą w odpowiedni sposób inicjalizować obiekty typu `Czołg`. Przy inicjalizacji musi być podany zawsze numer czołgu. Pozostałe parametry mają następujące wartości domyślne:

- nazwa czołgu: **Rudy**
- kaliber dział: **76,2**
- pozycja `x`: **0**
- pozycja `y`: **0**

Utworzenie szkieletu potrzebnych klas

Zacznijmy od utworzenia klas i struktur według wymagań. Przykładowa implementacja struktury Punkt może wyglądać następująco:

```
struct Punkt{
    private int x;
    private int y;

    public int PobierzX() {
        return x;
    }
    public int PobierzY() {
        return y;
    }
}
```

Klasę Działo możemy zdefiniować w następujący sposób:

```
class Działo {
    private double kaliber;
    public double PobierzKaliber() {
        return kaliber;
    }
}
```

Korzystając z powyższych klas, klasę Czołg możemy zaimplementować w następujący sposób:

```
class Czołg {
    private int numerCzołgu;
    private string nazwa;
    private Działo działo;
    private Punkt pozycja;

    public string PobierzInformacje() {
        return string.Format("Czołg nr {0} o nazwie {1}, kaliber działa {2} znajduje się w punkcie ({3}; {4})", numerCzołgu, nazwa, działo.PobierzKaliber(), pozycja.PobierzX(), pozycja.PobierzY());
    }
}
```

Zauważmy, że wszystkie pola w nowo zdefiniowanych typach są prywatne. Zdefiniowaliśmy metody, które w pewien sposób informują o stanie obiektów tych klas, nie mamy jednak możliwości modyfikacji wartości poszczególnych pól.

Utworzenie zestawu konstruktorów

W powyższych klasach nie tylko nie możemy modyfikować wartości pól, ale również nie możemy na razie ich zainicjalizować. By to umożliwić, musimy zdefiniować odpowiednie konstruktory. Konstruktor struktury Punkt może wyglądać następująco:

```
public Punkt(int x, int y){
    this.x = x;
    this.y = y;
}
```

Dla klasy Działo przykładowa implementacja konstruktora znajduje się poniżej:

```
public Działo(double kaliber) {
    this.kaliber = kaliber;
}
```

Dla klasy Czołg utworzymy kilka konstruktorów, które umożliwią tworzenie obiektu dla różnego zestawu parametrów.

Do klasy `Czolg` dodajmy publiczny konstruktor inicjalizujący wszystkie pola. W celu inicjalizacji pól typu `Dzialo` i `Punkt` wykorzystamy konstruktory wcześniej zdefiniowane dla tych typów.

```
public Czolg(int nrCzolgu, string nazwa, double kaliber,
             int pozycjaX, int pozycjaY){
    numerCzolgu = nrCzolgu;
    this.nazwa = nazwa;
    dzialo = new Dzialo(kaliber);
    pozycja = new Punkt(pozycjaX, pozycjaY);
}
```

Do klasy `Czolg` dodajmy kolejne konstruktory, które będą wykorzystywać konstruktor zdefiniowany powyżej w celu uniknięcia powielania kodu. Unikanie powielania kodu ułatwi późniejszą jego modyfikację i pielęgnację:

```
public Czolg(int nrCzolgu, string nazwa, Dzialo dzialo, Punkt punkt)
    : this(nrCzolgu, nazwa, dzialo.PobierzKaliber(),
          punkt.PobierzX(), punkt.PobierzY())
{}
public Czolg(int nrCzolgu, string nazwa, double kaliber)
    : this(nrCzolgu, nazwa, kaliber, 0, 0)
{}
public Czolg(int nrCzolgu, string nazwa)
    : this(nrCzolgu, nazwa, 76.2)
{}
public Czolg(int nrCzolgu)
    : this(nrCzolgu, "Rudy")
{}
}
```

Utworzenie programu testowego

Zaprezentujemy jak można skorzystać z wcześniej utworzonej klasy `Czolg`. Sprawdźmy również, czy możemy wywołać bezparametrowy konstruktor dla klasy `Dzialo` oraz struktury `Punkt`. Przykładowy kod testujący nasze klasy znajduje się poniżej:

```
static void Main(string[] args){
    Punkt p1 = new Punkt(); //wszystko OK - struktura
    //Dzialo dz = new Dzialo(); //Bład - dla klas nie jest definiowany
    //konstruktor domyslny gdy istnieje inny konstruktor
    Dzialo dz1 = new Dzialo(78);

    Czolg czolg1 = new Czolg(100, "Czolg 1", dz1, p1);
    Czolg czolg2 = new Czolg(102);
    Czolg czolg3 = new Czolg(103, "Czolg 3", 83.5, 10, 34);

    Console.WriteLine(czolg1.PobierzInformacje());
    Console.WriteLine(czolg2.PobierzInformacje());
    Console.WriteLine(czolg3.PobierzInformacje());
    Console.ReadKey();
}
```

Kopiowanie płytkie i głębokie

Naszym zadaniem jest dodanie do klasy `Czolg` metody `Klonuj`, która będzie wykonywać płytkie kopiowanie oraz konstruktora, do którego będzie przekazywana zmienna typu `Czolg`. Konstruktor powinien utworzyć nowy, niezależny obiekt (kopiowanie głębokie).

Dodanie metod pomocniczych

Zanim przejdziemy do głównego celu ćwiczenia, dodajmy metody, które umożliwią zmianę parametrów obiektu `Czolg`. Będą one wykorzystywane przy testowaniu skopiowanych obiektów.

Do struktury `Punkt` dodajmy dwie publiczne metody `ZmienX`, `ZmienY`:

```
public void ZmienX(int noweX){
    x = noweX;
}
public void ZmienY(int noweY){
    y = noweY;
}
```

Do klasy Dzialo dodajmy publiczną metodę ZmienKaliber:

```
public void ZmienKaliber(double nowyKaliber){
    kaliber = nowyKaliber;
}
```

Do klasy Czolg dodajmy publiczne metody ZmienKaliber, ZmienPozycje oraz ZmienNazwe:

```
public void ZmienKaliber(double nowyKaliber){
    dzialo.ZmienKaliber(nowyKaliber);
}

public void ZmienPozycje(int x,int y){
    pozycja.ZmienX(x);
    pozycja.ZmienY(y);
}

public void ZmienNazwe(string nowaNazwa){
    nazwa = nowaNazwa;
}
```

Utworzenie metod tworzących kopie obiektu

Do klasy Czolg dodamy metodę publiczną Klonuj. Ponieważ chcemy wykonać płytką kopię obiektu, żeby nie wywarzać już otwartych drzwi, użyjemy metody MemberwiseClone, którą posiada każdy typ w języku C#:

```
public Czolg Klonuj(){
    return (Czolg)this.MemberwiseClone();
}
```

Do klasy Czolg dodamy publiczny konstruktor, do którego będzie przekazywana zmienna typu Czolg. Konstruktor będzie wykonywał kopię głęboką. Wcześniej dodajmy do klasy Dzialo publiczny konstruktor, który przyjmuje jako parametr zmienną typu Dzialo:

```
class Dzialo{
    ...
    public Dzialo(Dzialo prototyp){
        kaliber = prototyp.kaliber;
    }
}

class Czolg{
    ...
    public Czolg(Czolg prototyp){
        numerCzolgu = prototyp.numerCzolgu;
        nazwa = prototyp.nazwa;
        dzialo = new Dzialo(prototyp.dzialo);
        pozycja = prototyp.pozycja;
    }
}
```

Zwróćmy uwagę, że w powyższym kodzie dla pola typu Punkt wykonujemy zwykłe przypisanie, ponieważ w przypadku typów wartościowych każda zmienna zawiera bezpośrednio dane. W

przypadku pola `dzialo`, które jest typu referencyjnego, samo przypisanie prowadziłoby jedynie do skopiowania referencji.

Utworzenie programu testowego

Utwórzmy kod za pomocą którego porównamy kopiowanie głębokie i kopiowanie płytke. Przykładowy kod znajduje się poniżej.

```
static void Main(string[] args){
    ...
    Console.WriteLine("\nKopiowanie płytke.");
    Czolg original1 = new Czolg(200, "Oryginal 1", 100, 10, 10);
    Czolg klon1 = original1.Klonuj();
    Console.WriteLine("Oryginał: {0}", original1.PobierzInformacje());
    Console.WriteLine("Klon: {0}", klon1.PobierzInformacje());
    Console.WriteLine("Zmieniamy klona: ");
    klon1.ZmienKaliber(300);
    klon1.ZmienNazwe("Klon 1");
    klon1.ZmienPozycje(55,55);
    Console.WriteLine("Oryginał: {0}", original1.PobierzInformacje());
    Console.WriteLine("Klon: {0}", klon1.PobierzInformacje());

    Console.WriteLine("\nKopiowanie głębokie.");
    Czolg original2 = new Czolg(200, "Oryginal 2", 100, 10, 10);
    Czolg klon2 = new Czolg(original2);
    Console.WriteLine("Oryginał: {0}", original2.PobierzInformacje());
    Console.WriteLine("Klon: {0}", klon2.PobierzInformacje());
    Console.WriteLine("Zmieniamy klona: ");
    klon2.ZmienKaliber(300);
    klon2.ZmienNazwe("Klon 2");
    klon2.ZmienPozycje(55,55);
    Console.WriteLine("Oryginał: {0}", original2.PobierzInformacje());
    Console.WriteLine("Klon: {0}", klon2.PobierzInformacje());
    ...
}
```

Skompiluj, uruchom i przetestuj działanie programu. Zwróć uwagę, że w przypadku kopiowania płytkiego zmiana parametru kaliber w kopii powoduje też zmianę tego parametru w oryginale. W przypadku kopiowania głębokiego kopia i oryginał są niezależne.

Gotowy powyższy projekt znajduje się w katalogu **Demo\Modul02**.

Porady praktyczne

- Pamiętaj, że w języku C# konstruktor jest to metoda wywoływana przy pomocy operatora `new`.
- Definicje wszystkich konstruktorów umieszczaj w jednym miejscu w klasie. Ułatwi to późniejszą pielęgnację kodu inicjalizującego. Definicję wszystkich konstruktorów możesz umieścić w dyrektywie `#region`, przypisując mu np. nazwę konstruktory.
- Konstruktor bezargumentowy (domyślny) dla klas jest automatycznie generowany pod warunkiem, że w danej klasie nie ma zdefiniowanego innego konstruktora.
- Nie zapomnij o modyfikatorze dostępu umieszczonym przed definicją konstruktora. Pominięcie modyfikatora dostępu powoduje, że konstruktor przez domyślność jest prywatny. Może to spowodować, że nie będziesz mógł utworzyć obiektów tej klasy.
- W przypadku struktur, konstruktor domyślny jest zawsze generowany, nie możesz jednak dostarczyć własnej wersji takiego konstruktora.
- Konstruktor automatycznie wygenerowany inicjalizuje wszystkie pola wartością zero odpowiednią dla danego typu.

- W przypadku klas, pola, którym w konstruktorze nie zostanie nadana wartość, będą zainicjalizowane wartością zero odpowiednią dla danego typu.
- Pamiętaj, że jeżeli piszesz własny konstruktor dla struktury, musisz w nim zainicjalizować wszystkie pola.
- Definiując kilka konstruktorów dla klasy lub struktury unikaj powtarzania kodu w poszczególnych konstruktorach. Zamiast tego możesz wywołać konstruktor z innego konstruktora przy pomocy listy inicjalizacyjnej oraz słowa kluczowego `this`. Ułatwi to pielęgnację kodu i ewentualne późniejsze jego modyfikacje. Unikniesz redundancji kodu.
- Pamiętaj, że w wywołaniu konstruktora przez inny konstruktor nie może dojść do rekurencji (ani bezpośredniej, ani pośredniej). Dobrze jest, jeśli się zdecydujesz, że konstruktor z mniejszą liczbą parametrów wywołuje konstruktor z liczbą parametrów o jeden większą, dostarczając domyślną wartość dla dodatkowego parametru. Dzięki takiej strategii możesz w łatwy sposób uniknąć sprzężeń zwrotnych (rekurencji).
- W przypadku gdy nie masz odpowiedniego konstruktora, możesz skorzystać z inicjalizatora obiektu. Pamiętaj, że przy pomocy inicjalizatora obiektu możesz nadać wartości tylko publicznym polom i właściwościom obiektu.
- Zanim zdecydujesz się zdefiniować destruktora, zastanów się, czy jest on na pewno konieczny. Definicja destruktora obniża wydajność procesu zwalniania pamięci. Sama potrzeba zwolnienia pamięci nie jest dobrym powodem do definiowania destruktora.
- W przypadku gdy zdecydujesz się utworzyć własny destruktora, rozważ implementację interfejsu `IDisposable` z metodą `Dispose`.
- W przypadku gdy utworzenie obiektu może być czynnością czasochłonną (konieczność skomplikowanych obliczeń, potrzeba połączenia z innym komputerem lub bazą danych), rozważ dostarczenie możliwości utworzenia obiektu z prototypu w celu poprawienia wydajności.
- Do zaimplementowania wzorca projektowego prototypu w języku C# możesz wykorzystać m.in. metodę `MemberwiseClone`, interfejs `ICloneable` lub mechanizm serializacji.
- Pamiętaj, że metoda `MemberwiseClone` wykonuje kopiowanie płytke, natomiast przy pomocy mechanizmu serializacji możesz wykonać kopiowanie głębokie.
- Do utworzenia obiektu w oparciu o obiekt już istniejący możesz również użyć konstruktora, do którego jako argument prześlesz zmienną typu zgodnego z tworzonym obiektem.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- wiesz, co to jest konstruktor
- znasz pojęcie destruktora
- umiesz definiować własne konstruktory
- wiesz, co to jest lista inicjalizacyjna konstruktora
- znasz różnice między inicjalizacją klas i struktur
- wiesz, co to są inicjalizatory obiektów i potrafisz z nich korzystać
- potrafisz wywołać konstruktor z innego konstruktora
- rozumiesz różnice między kopiowaniem płytkim a kopiowaniem głębokim i potrafisz je zaimplementować
- wiesz, do czego możesz wykorzystać metodę `MemberwiseClone`
- wiesz, do czego służy wzorec projektowy o nazwie prototyp

Dodatkowe źródła informacji

1. Daniel Solis, *Illustrated C# 2008*, Apress, 2008

Książka dla tych wszystkich którzy pragną nauczyć się tworzyć programy w języku C#. Zawiera dokładne omówienie tego języka.

2. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka dla programistów chcących nauczyć się programować w języku C#.

3. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Wzorce projektowe Wydanie II*, WNT, 2008

Książka, która wprowadziła pojęcie wzorców projektowych do informatyki. Lektura obowiązkowa dla każdego, kto chce poznać temat wzorców projektowych.

4. Steven John Metsker, *C#. Wzorce projektowe*, Helion, 2005

Książka jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia, jak wykorzystać cechy języka C# do tworzenia poprawnego kodu poprzez zastosowanie wzorców.

5. Judith Bishop, *C# 3.0 Design Patterns*, O'Reilly Media, Inc. 2008

Książka, podobnie jak poprzednia, jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia również nowe cechy języka C# 3.0.

6. Codeguru, <http://www.codeguru.pl>

Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie. Zwróć szczególnie uwagę na artykuły:
<http://www.codeguru.pl/article-580.aspx>; <http://www.codeguru.pl/article-597.aspx>; <http://www.codeguru.pl/article-649.aspx>

Laboratorium podstawowe

Problem 1 (czas realizacji 30 min)

Twoja firma opracowuje program kadrowy dla pewnej firmy. Twoim zadaniem jest stworzenie i przetestowanie klasy *Osoba*. O każdej osobie powinieneś posiadać następujące informacje:

- imię i nazwisko
- numerEwidencyjny
- adresZamieszkania. Klasa *Adres* zawiera następujące dane:
 - nazwaUlicy, miejscowosc
 - numerDomu
 - numerMieszkania

Zakładając, że najczęściej spotykanym nazwiskiem jest Kowalski, firma znajduje się w Warszawie i ma mieszkania pracownicze przy ulicy Aleje Jerozolimskie, stwórz odpowiednie konstruktory. Dodaj też możliwość kopiowania płytkiego i kopiowania głębokiego do klasy *Osoba*.

Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition	<ul style="list-style-type: none"> • Otwórz Visual C# 2008 Express Edition. • Z menu wybierz File -> New Project. • Z listy Visual Studio installed templates wybierz Console Application. • W polu Name wpisz Kadry. • Kliknij OK. • Z menu wybierz File -> Save Kadry. • W polu Location wybierz folder w którym będzie zapisany projekt. • Zaznacz pole wyboru Create directory for solution. • W polu Solution Name wpisz Modul02. • Naciśnij przycisk Save.
2. Dodaj do projektu klasę <i>Osoba</i>	<ul style="list-style-type: none"> • Z menu głównego wybierz Project -> Add Class. • W oknie dialogowym Add New Item – Kadry z listy Visual Studio installed templates wybierz Class. • W polu Name wpisz Osoba. • Kliknij OK.
3. Dodaj do projektu klasę <i>Adres</i> i ją zaimplementuj	<ul style="list-style-type: none"> • W pliku Osoba.cs w przestrzeni nazw Kadry tuż przed definicją klasy Osoba dodaj definicję klasy Adres według wymagań: <pre>class Adres{ private string miejscowosc; private string kod; private string nazwaUlicy; private int numerDomu; private int? numerMieszkania; }</pre> • Do klasy Adres dodaj odpowiednie konstruktory: <ul style="list-style-type: none"> – konstruktor, który przyjmuje argumenty dla wszystkich pól klasy adres – konstruktor, który w odróżnieniu od powyższego nadaje polu miejscowosc wartość Warszawa, a polu kod wartość 02-222 – konstruktor, który w odróżnieniu od powyższego nadaje polu nazwaUlicy wartość Aleje Jerozolimskie – konstruktor, który w odróżnieniu od powyższego nadaje polu

numerMieszkania wartość null

- konstruktor, który jako argument przyjmuje zmienną typu **Adres** i tworzy jego kopię

```
public Adres(int numerDomu, int? numerMieszkania,
    string nazwaUlicy, string kod, string miejscowosc){
    this.numerDomu = numerDomu;
    this.numerMieszkania = numerMieszkania;
    this.nazwaUlicy = nazwaUlicy;
    this.kod = kod;
    this.miejscowosc = miejscowosc;
}
```

```
public Adres(int numerDomu, int? numerMieszkania,
    string nazwaUlicy)
    :this(numerDomu,numerMieszkania,nazwaUlicy,"02-222", "Warszawa")
{}
```

```
public Adres(int numerDomu, int? numerMieszkania)
    : this(numerDomu, numerMieszkania, "Aleje Jerozolimskie")
{}
```

```
public Adres(int numerDomu)
    :this(numerDomu, null)
{}
```

```
public Adres(Adres adres)
    : this(adres.numerDomu, adres.numerMieszkania, adres.nazwaUlicy,
        adres.kod,adres.miejscowosc)
{}
```

- Do klasy **Adres** dodaj publiczną metodę zwracającą informacje adresowe w formie napisu:

```
public string ZwrocInformacjeAdresowe(){
    return numerMieszkania != null ?
        string.Format("{0} {1} ul. {2} {3} m. {4}",
            kod, miejscowosc, nazwaUlicy, numerDomu, numerMieszkania) :
        string.Format("{0} {1} ul. {2} {3}",
            kod, miejscowosc, nazwaUlicy, numerDomu);
}
```

- Do klasy **Adres** dodaj publiczne metody zwracającą pojedyncze informacje adresowe:

```
public string ZwrocMiejscowosc(){
    return miejscowosc;
}
```

```
public string ZwrocNazweUlicy(){
    return nazwaUlicy;
}
```



```
public string ZwrocKod(){
    return kod;
}
```

```
public int ZwrocNumerDomu(){
    return numerDomu;
}
```

```
public int? ZwrocNumerMieszkania(){
    return numerMieszkania;
}
```


	<ul style="list-style-type: none"> Do klasy Adres dodaj publiczną metodę zmieniającą dane adresowe: <pre> public void ZmienAdres(){ Console.Write("Podaj nazwę miejscowości: "); miejscowosc = Console.ReadLine(); Console.Write("Podaj kod: "); kod = Console.ReadLine(); Console.Write("Podaj nazwę ulicy: "); nazwaUlicy = Console.ReadLine(); do{ Console.Write("Podaj numer domu: "); } while (!int.TryParse(Console.ReadLine(), out numerDomu)); Console.Write("Czy jest numer mieszkania <t/n>: "); char c = Console.ReadKey().KeyChar; if (c == 't'){ Console.WriteLine(); int i; do{ Console.Write("Podaj numer mieszkania: "); } while (!int.TryParse(Console.ReadLine(), out i)); numerMieszkania = i; } else{ numerMieszkania = null; } } </pre>
4. Dodaj implementację do klasy Osoba	<ul style="list-style-type: none"> Do klasy Osoba dodaj następujące pola: <pre> private string nazwisko; private string imie; private int numerEwidencyjny; private Adres adresZamieszkania; </pre> Do klasy Osoba dodaj odpowiednie konstruktory: <ul style="list-style-type: none"> konstruktory, które przyjmują argumenty dla wszystkich pól klasy Osoba (jeden, który przyjmuje jako jeden z argumentów zmienną klasy adres oraz drugi, który przyjmuje parametry adresu jako oddzielne argumenty) konstruktor, który przyjmuje następujące argumenty: imie, numerEwidencyjny i numerDomu (pozostałym polom zostają nadane wartości domyślne) konstruktor, który przyjmuje jako argument parametr typu Osoba <pre> public Osoba(int numerEwidencyjny, string imie, string nazwisko, int numerDomu, int? numerMieszkania, string nazwaUlicy, string kod, string miejscowosc){ this.numerEwidencyjny = numerEwidencyjny; this.imie = imie; this.nazwisko = nazwisko; adresZamieszkania = new Adres(numerDomu, numerMieszkania, nazwaUlicy, kod, miejscowosc); } public Osoba(int numerEwidencyjny, string imie, string nazwisko, Adres adres) : this(numerEwidencyjny, imie, nazwisko, adres.ZwrocNumerDomu(), adres.ZwrocNumerMieszkania(), adres.ZwrocNazweUlicy(), </pre>


	<pre> adres.ZwrocKod(), adres.ZwrocMiejscowosc()) {} public Osoba(int numerEwidencyjny, string imie, int numerDomu) : this(numerEwidencyjny, imie, "Kowalski", numerDomu, null, "Aleje Jerozolimskie", "02-222", "Warszawa") {} public Osoba(Osoba osoba){ numerEwidencyjny = osoba.numerEwidencyjny; imie = osoba.imie; nazwisko = osoba.nazwisko; adresZamieszkania = new Adres(osoba.adresZamieszkania); } </pre> <ul style="list-style-type: none"> Do klasy Osoba dodaj publiczną metodę zwracającą informacje szczegółowe na temat danej osoby w formie napisu: <pre> public string ZwrocInformacjeOsobowe(){ return string.Format("Pan(i) {0} {1} o numerze ewidencyjnym {2} zamieszkały(a): {3}", imie, nazwisko, numerEwidencyjny, adresZamieszkania.ZwrocInformacjeAdresowe()); } </pre> Do klasy Osoba dodaj publiczną metodę Klonuj zwracającą płytką kopię obiektu: <pre> public Osoba Klonuj(){ return (Osoba)this.MemberwiseClone(); } </pre> Do klasy Osoba dodaj publiczną metodę zmieniającą dane osobowe: <pre> public void ZmienDaneOsobowe(){ Console.Write("Podaj imię: "); imie = Console.ReadLine(); Console.Write("Podaj nazwisko: "); nazwisko = Console.ReadLine(); do{ Console.Write("Podaj numer ewidencyjny: "); } while(!int.TryParse(Console.ReadLine(), out numerEwidencyjny)); adresZamieszkania.ZmienAdres(); } </pre>
5. Przetestuj klasę Osoba	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Do metody Main dodaj kod, który przetestuje, czy klasa Osoba działa prawidłowo. Przykładowy kod jest zamieszczony poniżej: <pre> Osoba os1 = new Osoba(1, "Jan", "Nowak", 23, 12, "Kwiatowa", "97-300", "Piotrków Tryb"); Osoba klon1 = os1.Klonuj(); Console.WriteLine("Oryginał {0}", os1.ZwrocInformacjeOsobowe()); Console.WriteLine("Klon: {0}", klon1.ZwrocInformacjeOsobowe()); Console.WriteLine("\n***Zmieniamy klona:**\n"); klon1.ZmienDaneOsobowe(); Console.WriteLine("Oryginał: {0}", os1.ZwrocInformacjeOsobowe()); Console.WriteLine("Klon: {0}", klon1.ZwrocInformacjeOsobowe()); Console.WriteLine("\n***Kopiowanie głębokie**\n"); Adres adr = new Adres(13); Osoba os2 = new Osoba(10, "Jacek", "Wiśniewski", adr); Osoba klon2 = new Osoba(os2); </pre>

	<pre> Console.WriteLine("\nOryginal drugi: {0}", os2.ZwrocInformacjeOsobowe()); Console.WriteLine("Klon drugi: {0}", klon2.ZwrocInformacjeOsobowe()); Console.WriteLine("\n***Zmieniamy drugiego klona:***\n"); klon2.ZmienDaneOsobowe(); Console.WriteLine("Oryginal drugi: {0}", os2.ZwrocInformacjeOsobowe()); Console.WriteLine("Klon drugi: {0}", klon2.ZwrocInformacjeOsobowe()); Console.ReadKey(); </pre>
6. Skompiluj, a następnie uruchom program	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. W celu uruchomienia programu, z menu Debug wybierz Start Debugging. <p> Które parametry osoby zmieniają się w oryginale w przypadku zmiany klonu przy kopiowaniu płytkim. Dlaczego? Czy w przypadku kopiowania głębokiego kopia i oryginał są zupełnie niezależne?</p>
7. Sprawdź, czy możesz korzystać z konstruktora domyślnego klasy Adres	<ul style="list-style-type: none"> Na koniec metody Main dodaj następujący kod: <pre>Adres adr2 = new Adres();</pre> Spróbuj skompilować projekt. <p> Jaki błąd zgłaszany jest przez kompilator? Czy kod kompilowałby się bezbłędnie, jeżeli Adres byłby strukturą, a nie klasą?</p> <ul style="list-style-type: none"> Skasuj lub umieść w komentarzu dodaną ostatnio linię.

Problem 2 (czas realizacji 15 min)

Masz napisaną klasę **Lista**. Klasa ta implementuje listę jednokierunkową. Twoim zadaniem jest dodanie możliwości tworzenia głębokiej kopii listy. Kopie masz utworzyć zarówno rekurencyjnie, jak i iteracyjnie. Struktura lista jednokierunkowa jest mówiona w kursie „Wprowadzenie do programowania”.

Zadanie	Tok postępowania
1. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestListy. Kliknij OK.
2. Zaznacz projekt TestKolejki jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestListy, a następnie z menu kontekstowego wybierz Set as StartUp Project.
3. Do projektu TestListy dodaj plik z implementacją listy jednokierunkowej	<ul style="list-style-type: none"> Z menu wybierz Project ->Add Existing Item.  Zwróć uwagę, aby przed wykonaniem powyższego polecenia w okienku Solution Explorer był zaznaczony projekt TestListy. W oknie dialogowym Add ExistingItem – TestListy wybierz plik Listy.cs, który znajduje się w katalogu Kurs\Lab\Start\Modul02, gdzie Kurs jest

	<p>katalogiem, gdzie zainstalowano pliki kursu.</p> <ul style="list-style-type: none"> Kliknij Add.
<p>4. Dodaj do klasy List możliwość rekurencyjnego tworzenia głębokiej kopii listy</p>	<ul style="list-style-type: none"> Przejdź do pliku List.cs. Do klasy Wezel dodaj publiczną metodę KlonujRekurencyjnie. Implementacja jej powinna mieć następującą postać: <pre>public void KlonujRekurencyjnie(Wezel wezel){ wezel.Dane = Dane; if (Nastepny != null){ wezel.Nastepny = new Wezel(); Nastepny.KlonujRekurencyjnie(wezel.Nastepny); } }</pre> Do klasy Lista dodaj publiczną metodę KlonujRekurencyjnie. Implementacja jej powinna mieć następującą postać: <pre>public Lista KlonujRekurencyjnie(){ Lista nowa = new Lista(); if (glowa != null){ nowa.glowa = new Wezel(); glowa.KlonujRekurencyjnie(nowa.glowa); } return nowa; }</pre>
<p>5. Dodaj do klasy List możliwość iteracyjnego tworzenia głębokiej kopii listy</p>	<ul style="list-style-type: none"> Do klasy Lista dodaj publiczną metodę KlonujIteracyjnie. Implementacja jej powinna mieć następującą postać: <pre>public Lista KlonujIteracyjnie(){ Lista nowa = new Lista(); if (glowa != null){ nowa.glowa = new Wezel(); Wezel tmp = glowa; Wezel tmp2 = nowa.glowa; tmp2.Dane = glowa.Dane; while (tmp.Nastepny != null){ tmp2.Nastepny = new Wezel(); tmp2 = tmp2.Nastepny; tmp = tmp.Nastepny; tmp2.Dane = tmp.Dane; } } return nowa; }</pre> <p> Zastanów się, co należałoby zmienić w powyższym kodzie, jeżeli na liście nie byłyby przechowywane napisy, tylko np. obiekty klasy Osoba? Chcesz oczywiście zachować pełną niezależność obu list.</p>
<p>6. Przetestuj obie metody klonowania obiektów klasy List</p>	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Do metody Main dodaj kod, który przetestuje obie metody kopiujące obiekty klasy Lista. Kod metody Main może wyglądać następująco: <pre>Listy.Lista oryginal = new Listy.Lista(); oryginal.DodajDoGlowy("Ania"); oryginal.DodajDoGlowy("Agnieszka"); oryginal.DodajDoGlowy("Wiktoria"); oryginal.DodajDoGlowy("Kasia"); Listy.Lista kopiaR = oryginal.KlonujRekurencyjnie(); Listy.Lista kopiaI = oryginal.KlonujIteracyjnie(); Console.WriteLine("Wypisujemy przed modyfikacjami: ");</pre>

	<pre> Console.WriteLine("\n***Oryginał: "); oryginal.WypiszWszystko(); Console.WriteLine("\n***Kopia R: "); kopiaR.WypiszWszystko(); Console.WriteLine("\n***Kopia I: "); kopiaI.WypiszWszystko(); Console.WriteLine("\nModyfikujemy:"); Console.ReadKey(); kopiaR.DodajDoGlowy("Zosia"); kopiaR.DodajDoGlowy("Jola"); kopiaI.UsunZGlowy(); kopiaI.UsunZGlowy(); Console.WriteLine("\n***Oryginał: "); oryginal.WypiszWszystko(); Console.WriteLine("\n***Kopia R: "); kopiaR.WypiszWszystko(); Console.WriteLine("\n***Kopia I: "); kopiaI.WypiszWszystko(); Console.ReadKey(); </pre>
7. Skompiluj i uruchom program	<ul style="list-style-type: none"> • Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. • W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 90 min)

Masz napisać program, który zawiera bazę danych osób. W celu poprawienia wydajności wyszukiwania osób wybrano drzewo binarne jako strukturę danych, w której będą przechowywane rekordy. Masz już szkielet aplikacji. Pliki startowe do tego zadania znajdują się w katalogu **Kurs\Lab\Start\Modul02**, gdzie **Kurs** jest katalogiem, w którym zainstalowano pliki kursu. Są to:

- **Osoba.cs** – znajduje się tu implementacja klasy **Osoba**
- **Drzewa.cs** – implementacja drzewa przechowującego osoby

Drzewo binarne jest omówione w kursie „Wprowadzenie do programowania”.

Twoim zadaniem jest:

- Dodanie konstruktora do struktury **Data** (konstruktor przyjmuje trzy argumenty: rok, miesiąc, dzień). Sprawdź poprawność wprowadzanych danych.
- Dodanie konstruktorów do klasy **Osoba**:
 - Konstruktor bezargumentowego, który inicjalizuje wszystkie pola wartościami domyślnymi (Kowalski, Jan, 1990,12,1).
 - Konstruktor przyjmującego argumenty: imię, nazwisko, rok, miesiąc, dzień.
- Dodanie konstruktorów do klasy **Wezeł**:
 - Konstruktor bezargumentowego, inicjalizującego pole **Dane** przy pomocy domyślnego konstruktora klasy **Osoba**, a pola **Lewy** i **Prawy** wartościami **null**.
 - Konstruktor przyjmującego jako argument zmienną typu **Osoba**. Pola **Lewy** i **Prawy** są inicjalizowane wartościami **null**.
 - Konstruktor przyjmującego argumenty: imię, nazwisko, rok, miesiąc, dzień. Pola **Lewy** i **Prawy** są inicjalizowane wartościami **null**.
- Dodanie konstruktorów do klasy **Drzewo**.
 - Konstruktor bezparametrowego inicjalizujące pole **korzen** wartością **null**.
 - Konstruktor przyjmującego argumenty: imię, nazwisko, rok, miesiąc, dzień. Konstruktor ten tworzy obiekt **Osoba** według podanych parametrów i dodaje go jako pierwszy element drzewa.
 - Konstruktor przyjmującego jako argument tablicę obiektów klasy **Osoba**. Konstruktor ten dodaje wszystkie elementy tablicy do tworzonego drzewa.
- Dodanie możliwości tworzenia drzewa w oparciu o istniejący obiekt drzewa. Ma być wykonana kopia głęboka drzewa (możesz zastosować podobny algorytm, jak przy tworzeniu kopi listy metodą rekurencyjną, który został pokazany w laboratorium podstawowym w zadaniu drugim). Napisz dwie metody:
 - Pierwsza wykonuje kopię głęboką. Również obiekty klasy **Osoba** przechowywane w drzewie oryginalnym i kopii drzewa są różne.
 - Druga tworzy kopię drzewa, węzły w obu drzewach są niezależne, ale pola **Dane** w odpowiadających sobie węzłach odwołują się do tego samego obiektu typu **Osoba**.
- Dodanie możliwości kopiowania płytkiego obiektu drzewa.

Napisz również program, który sprawdzi poprawność tworzenia kopii drzewa.

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 3

Wersja 2

Właściwości i indeksatory

Spis treści

Właściwości i indeksatory	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	8
Porady praktyczne	11
Uwagi dla studenta	12
Dodatkowe źródła informacji	12
Laboratorium podstawowe	14
Problem 1 (czas realizacji 35 minut)	14
Problem 2 (czas realizacji 10 minut)	19
Laboratorium rozszerzone	22
Zadanie 1 (czas realizacji 90 min)	22
Zadanie 2 (czas realizacji 45 min)	22

Informacje o module

Opis modułu

W tym module poznasz co to są właściwości i indeksatory oraz nauczysz się nimi posługiwać w języku C#. Dowiesz się również do czego służą właściwości automatyczne. Dodatkowo nauczysz się, co to są pola tylko do odczytu i jak je definiować. Zostanie tu również przedstawiony wzorzec projektowy o nazwie proxy i jak przy jego pomocy uniezależnić definicję klasy od biblioteki wejścia-wyjścia.

Cel modułu

Celem modułu jest pokazanie możliwości wykorzystania właściwości i indeksatorów przy definicji klasy w języku C#.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- znał pojęcie właściwości i potrafił je definiować
- znał pojęcie indeksatora
- wiedział, co to są pola tylko do odczytu
- rozumiał potrzebę stosowania enkapsulacji
- wiedział, co to jest i do czego służy wzorzec projektowy proxy

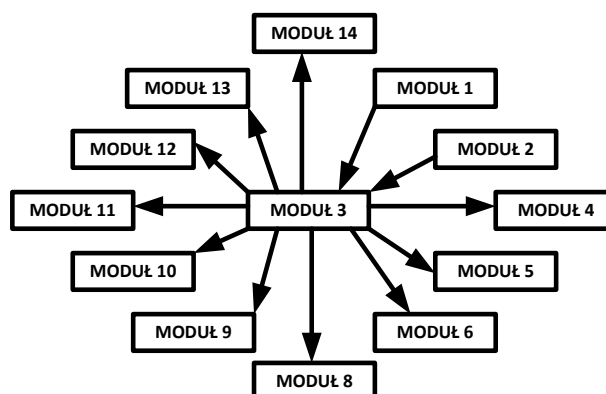
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinieneś:

- potrafić definiować klasę
- wiedzieć, co to jest pole i co to jest metoda
- znać modyfikatory dostępu (public i private)
- rozumiał pojęcie wzorców projektowych (patrz: moduł „Konstruktor”)

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w module „Pojęcie klasy” oraz „Konstruktor”.



Rys. 5 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Jesteś już w miarę doświadczonym programistą. Uczestniczyłeś w pewnym projekcie, gdzie stworzyłeś klasę `Student`. Jedną z cech studenta jest numer indeksu. Numer indeksu zaimplementowałeś jako liczbę całkowitą, ponieważ w czasie analizy zauważyłeś, że kolejnym studentom są przydzielone kolejne liczby całkowite. Twoja klasa `Student` jest wykorzystywana w kilku programach wspomagających pracę w dziekanatach na różnych wydziałach. Do tej pory każdy dziekanat prowadził własną numerację studentów. Władzę uczelni postanowili jednak ujednolicić numerację indeksów na wszystkich wydziałach. Zarządziły że numer indeksu oprócz liczby będzie się składał dodatkowo z litery, która będzie symbolizować wydział. Prosta zmiana typu pola z typu liczbowego na typ `string` spowoduje, że programy odwołujące się bezpośrednio do pola reprezentującego numer indeksu przestaną działać. Rodzi się więc pytanie, czy można napisać tak program, aby późniejsze zmiany typu pól można było przeprowadzać w miarę szybko i bezboleśnie. Odpowiedzią jest enkapsulacja, czyli ukrycie pól i umożliwienie dostępu do nich tylko przy użyciu metod danej klasy. Wtedy problem z numerem indeksu możesz rozwiązać modyfikując tylko klasę `Student`, bez konieczności zmiany programów z niej korzystających. Zakładając, że stosowałeś enkapsulację, musiałeś mieć metodę, która zwraca numer indeksu w postaci liczby całkowitej. Przy zmianie typu pola reprezentującego numer indeksu, będziesz musiał również zmienić implementację metody zwracającej numer indeksu. Nowa implementacja może usunąć ostatni znak reprezentujący wydział, następnie przekonwertować uzyskany napis na liczbę i przekazać ją do programu. Oczywiście powinieneś również dodać metodę zwracającą numer indeksu jako napis. Będą z niej korzystać wszystkie programy, którym potrzebny jest ujednolicony sposób numeracji indeksów.

Nienaturalne wydaje się jednak posługiwanie się metodą tam, gdzie spodziewasz się pola. W tym rozdziale zostanie opisany mechanizm języka C#, który umożliwia enkapsulację, a zarazem odwoływanie się do pól, oczywiście w sposób pośredni, tak jakbyś odwoływał się do nich bezpośrednio. Brzmi to zawile. Zapoznaj się z dalszą częścią tego rozdziału.

Tworząc program tworzysz różne kolekcje. Raz wystarczy umieścić obiekty w zwykłej tablicy, za drugim razem opakowujesz kolekcję obiektów w specjalną klasę. Do elementu tablicy można odwołać się przy pomocy indeksu. W języku C# możesz zdefiniować tak klasę, że jeżeli jej obiekty zawierają pewną kolekcję, to do elementów tej kolekcji możesz odwoływać się przy pomocy indeksu. Co więcej, indeks ten nie koniecznie musi być nieujemną liczbą całkowitą. Szczegóły już za chwilę.

Podstawy teoretyczne

W języku C# istnieje konstrukcja, która łączy w sobie pole i metodę. Korzystamy z tego mechanizmu tak, jak by było to zwykłe pole, natomiast tak naprawdę wywołujemy metodę. Konstrukcję tą nazywamy *właściwością* i definiujemy w następujący sposób:

```
class Nazwa_Klasy {  
    ...  
    [modyfikator_dostępu] typ_właściwości nazwa_właściwości {  
        get{  
            //ciało metody get  
        }  
        set{  
            //ciało metody set  
        }  
    }  
}
```

Definiując właściwość określamy dwie metody: `set` i `get`. W powyższej definicji można zauważyć, że definicja tych metod jest specyficzna: nie zawiera wartości zwracanej ani argumentów. Musimy jednak pamiętać, że metoda `get` przekazuje do programu wartość, której typ jest zgodny z typem właściwości, czyli każda ścieżka wykonania musi kończyć się instrukcją `return`. Metoda `set` natomiast nie zwraca żadnej wartości. Przyjmuje ona jeden argument o nazwie *value*, który jest przesyłany przez wartość. Typ tego argumentu jest zgodny z typem właściwości. Rozważmy to na przykładzie klasy `Student` i pola, które reprezentuje numer indeksu:

```
class Student {
    ...
    private int numerIndeksu;
    public int NumerIndeksu {    //definicja właściwości
        get { return numerIndeksu; }
        set { numerIndeksu = value; }
    }
}
```

W powyższym przykładzie zastosowano standardową konwencję języka C#. Nazwa pola oraz nazwa właściwości, przy pomocy której uzyskujemy dostęp do tego pola, różnią się tylko wielkością pierwszej litery.

Metoda `set` jest wywoływany wtedy, gdy próbujemy nadać wartość właściwości, np.:

```
Student s = new Student();
s.NumerIndeksu = 1234;    //argument value przyjmuje wartość 1234
```

Metoda `get` jest wywoływana wtedy, gdy próbujemy pobrać wartość reprezentowaną przez właściwość np.:

```
int nr = s.NumerIndeksu;
Console.WriteLine(s.NumerIndeksu);
```

Definiując właściwość możemy określić tylko jedną z metod. Implementując tylko metodę `get` tworzymy właściwość tylko do odczytu:

```
class Osoba {
    ...
    private int rokUrodzenia;
    public int Wiek {
        get { return DateTime.Now.Year - rokUrodzenia; }
    }
}
```

Podobnie, dodając samą metodę `set` tworzymy właściwość tylko do zapisu.

Metody `get` i `set` mogą mieć również różne modyfikatory dostępu. Implementujemy to w następujący sposób:

```
public typ_właściwości nazwa_właściwości {
    get {                                //metoda get jest publiczna (jak właściwość)
        //ciało metody get
    }
    private set {                        //prywatna metoda set
        //ciało metody set
    }
}
```

Podsumowując, właściwości powinniśmy definiować w następujących przypadkach:

- Pojęcie reprezentuje pewien atrybut (cechę) klasy lub obiektu, ale w celu obliczenia jego wartości trzeba wywołać pewne funkcje. Weźmy pod uwagę strukturę `DateTime`. Posiada ona właściwość `Now` (Teraz). W celu pobrania aktualnej daty i czasu musimy wywołać

odpowiednie funkcje systemowe. Dla nas Now (Teraz) jest zbiorem parametrów (aktualnym rokiem, godziną sekundą itp.), a nie wywołaniem metody. Podobnie w jednym z wcześniejszych przykładów zdefiniowaliśmy właściwość `Wiek`. W celu obliczenia wieku musimy wykonać pewne obliczenia, czyli tak naprawdę napisać metodę. Dla nas `wiek` jest jednak bardziej związany z atrybutami klasy, a nie jej funkcjonalnością.

- Jedną wartość można w prosty sposób obliczyć na podstawie drugiej. Wyobraźmy sobie, że mamy klasę `Towar`. Każdy towar posiada cenę netto oraz brutto. Wystarczy utworzyć pole reprezentującą tylko cenę netto. Do reprezentacji ceny brutto możemy utworzyć odpowiednią właściwość.
- Dostęp do pola klasy musi być kontrolowany w celu zapewnienia spójności (poprawności) obiektu.
- Ukrycie implementacji (typu). Problem ten został opisany w sekcji „Przykładowy problem” tego modułu. Dzięki temu, że dostęp do pola uzyskujemy przy pomocy właściwości, późniejsza zmiana typu pola nie wpływa na programy korzystające z danej klasy. Wystarczy tylko zmienić odpowiednio ciała metody `get` oraz `set`. W metodach tych należy dokonać odpowiedniej konwersji typu.

W wersji języka C# 3.0 wprowadzono *automatyczne właściwości* (ang. *Automatic Properties*). Ułatwiają one definicję par pole-właściwość. Zamiast pisać:

```
class NazwaKlasy {
    private typ_pola nazwaPola;
    public typ_pola NazwaPola    //definicja właściwości
    {
        set { nazwaPola = value; }
        get { return nazwaPola; }
    }
}
```

możemy zapisać krócej i prościej:

```
class NazwaKlasy {
    public typ_pola NazwaPola {    //definicja właściwości
        set; get;
    }
}
```

Powyższa definicja tworzy właściwość o nazwie `NazwaPola`, przy pomocy której mamy dostęp do anonimowego pola. Ponieważ utworzone automatycznie pole jest anonimowe, nie możemy się do niego odwoływać bezpośrednio – musimy zawsze używać właściwości. W przypadku automatycznych właściwości musimy podać obie metody (`set` i `get`) i obie nie mogą mieć definicji.

Używając właściwości do dostępu do pola możemy mieć obiekty co do wydajności. Wiadomo, że wywołanie metody jest obciążone dodatkowym narzutem. Należy tutaj jednak przypomnieć pojęcie *metod inline*, omówione w kursie „Wprowadzenie do programowania”. W momencie kompilacji, kompilator może wstawić w miejsce wywołania funkcji kod ją definiujący. Warto zwrócić uwagę, że w przypadku .NET Framework optymalizacji tej dokonuje kompilator bezpośredni (kompilujący kod pośredni na kod natywny), a nie kompilator języka C#.

Indeksatory

Zazwyczaj gdy klasa reprezentuje pewną kolekcję, chcielibyśmy mieć możliwość odwoływania się do poszczególnych jej elementów przy pomocy indeksu, podobnie jak robimy to w przypadku tablic. W języku C# możemy tego dokonać przy pomocy *indeksatorów* nazywanych też *indekserami* (ang. *indexers*). Indeksatory definiujemy przy pomocy słowa kluczowego `this` w następujący sposób:

```
[modyfikator_dostępu] typ_indeksatora this[<lista_argumentów>] {
    get{
```

```

        //ciało metody get
    }
    set{
        //ciało metody set
    }
}

<Lista_argumentów>::=
    nazwa_typu nazwa_argumentu[,...n]

```

Przeglądając powyższą definicję możemy zauważyć, że indeksatory definiujemy w bardzo podobny sposób do właściwości. Do metody `get` jednak przekazywane są argumenty, które umieściliśmy w nawiasach kwadratowych. Do metody `set`, podobnie jak w przypadku właściwości, przekazujemy argument o nazwie `value`, którego typ jest zgodny z typem indeksatora, ale dodatkowo przekazywane są również argumenty umieszczone w nawiasach kwadratowych. Mając zdefiniowaną klasę i utworzony obiekt tej klasy,

```

class NazwaKlasy {
    public double this[string s, int i]    //definicja indeksatora
        set {...}
        get {...}
}
...
NazwaKlasy nazwaZmiennej = new NazwaKlasy();

```

z indeksatora korzystamy w następujący sposób:

```

nazwaZmiennej["Argument1", 10] = 23.4;           //wywołanie metoda set,
                                                    //argument value równy 23.4

double x = nazwaZmiennej["Argument1", 10];       //wywołanie metody get
Console.WriteLine(nazwaZmiennej["Argument1", 10]); //wywołanie metoda get

```

Pole tylko do odczytu

Tworząc pole i właściwość tylko do odczytu skojarzoną z tym polem, nie zagwarantujemy, że nie będzie można zmienić wartości tego pola. W języku C# istnieje słowo kluczowe *readonly*, które umożliwia utworzenie pola, któremu wartość możemy nadać tylko w konstruktorze lub w miejscu definicji tego pola. Pole takie nazywamy polami tylko do odczytu i definiujemy je w następujący sposób.

```
readonly [modyfikator_dostępu] typ_pola nazwa_pola;
```

Wzorzec projektowy proxy

W poprzednim module mogliśmy zapoznać się z pojęciem prototypu. Teraz spróbujemy przyrzeć się kolejnemu wzorcowi projektowemu, a mianowicie wzorcowi *proxy*. Inną nazwą tego wzorca jest *surrogate*, co w języku polskim często tłumaczone jest jako *pełnomocnik*. Wzorzec ten należy do wzorców strukturalnych. Głównym jego celem jest dostarczenie obiektu, który kontroluje tworzenie i dostęp do innego obiektu.

Wzorzec ten implementuje się najczęściej w ten sposób, że każdy obiekt *proxy* utrzymuje jako pole referencję do żądanego obiektu i posiada metody, przy pomocy których możemy wywołać metody żądanego obiektu.

W książce „Elementy oprogramowania obiektowego wielokrotnego użytku” napisanej przez Ericha Gamme, Richarda Helma, Ralpha Johnsona i Johna Vlissidesa autorzy wyróżniają następujące rodzaje wzorców *proxy*:

- **Pełnomocnik zdalny** (ang. *remote proxy*) – służy do reprezentacji obiektu, który znajduje się np. na innym komputerze. Klienci lokalni za pośrednictwem obiektu *proxy* mogą korzystać ze

zdalnego obiektu, tak jakby on znajdował się na komputerze lokalnym. Wzorec ten powszechnie używany jest przy korzystaniu z usług Web Services.

- **Pełnomocnik wirtualny** (ang. *virtual proxy*) – zastępuje obiekt o dużych wymaganiach np. co do zajętości pamięci lub gdy utworzenie obiektu jest czasochłonne. Utworzenie obiektu jest odkładane w czasie do momentu, aż będzie konieczne, a część jego zobowiązań przejmuje na siebie obiekt proxy.
- **Pełnomocnik ochraniający** (ang. *protection proxy*) – pośredniczy w dostępie do obiektu. Obiekt nigdy nie jest bezpośrednio dostępny dla klientów; w jego imieniu występuje obiekt proxy, który określa, komu i jakie operacje oferowane przez obiekt można udostępnić.
- **Sprytny pełnomocnik** (ang. *smart proxy*) – używany jest wtedy, gdy przy wywołaniu metod danego obiektu konieczne jest wykonanie dodatkowych operacji.

My wprowadzimy jeszcze jednego pełnomocnika. Będzie on odpowiedzialny za wyświetlanie stanu obiektu oraz zmianę wartości właściwości i pól danego obiektu przy pomocy konkretnej biblioteki wejścia-wyjścia. Tworząc klasę nie jesteśmy w stanie przewidzieć, gdzie w przyszłości będzie ona wykorzystywana. Może być używana w programach konsolowych, jak również w aplikacjach okienkowych. Przy pomocy właściwości tej klasy, tworzymy warstwę, którą używamy do uzyskania dostępu do informacji zawartej w obiektach tej klasy. We właściwościach nie powinniśmy jednak używać operacji wejścia-wyjścia. Operacje te powinny być przesunięte do oddzielnej klasy, np.:

```
class Osoba {
    public string Imie { get; set; }
    public string Nazwisko { get; set; }
    private int rokUrodzenia;
    public int Wiek {
        get { return DateTime.Now.Year - rokUrodzenia; }
    }
    public Osoba(string imie, string nazwisko, int rok) {
        Imie = imie;
        Nazwisko = nazwisko;
        rokUrodzenia = rok;
    }
}

class OsobaDlaKonsoli {
    private Osoba osoba;
    public OsobaDlaKonsoli() {
        Console.Write("Podaj nazwisko: ");
        string nazwisko = Console.ReadLine();
        Console.Write("Podaj imię: ");
        string imie = Console.ReadLine();
        Console.Write("Podaj rok urodzenia: ");
        int rok = int.Parse(Console.ReadLine());
        osoba = new Osoba(imie, nazwisko, rok);
    }
    public void ZmienImie() {
        Console.Write("Podaj imię: ");
        osoba.Imie = Console.ReadLine();
    }
    public void ZmienNazwisko() {
        Console.Write("Podaj nazwisko: ");
        osoba.Nazwisko = Console.ReadLine();
    }
    public void WypiszOsobe() {
        Console.WriteLine("Pan(i) {0} {1} lat {2}",
            osoba.Imie, osoba.Nazwisko, osoba.Wiek);
    }
}
```

Przykładowe rozwiązanie

Definiowanie właściwości

W programie posługujemy się pojęciem pola powierzchni. Niestety w różnych częściach programu pole powierzchni jest liczone w różnych jednostkach. Aby ułatwić połączenie różnych części programu, możemy utworzyć następującą strukturę:

```
public struct PolePowierzchni {  
    public double MetrKwadratowy { set; get; }  
    public double Ar {  
        set { MetrKwadratowy = 100 * value; }  
        get { return MetrKwadratowy / 100; }  
    }  
    public double Hektar {  
        set { MetrKwadratowy = 10000 * value; }  
        get { return MetrKwadratowy / 10000; }  
    }  
}
```

Dzięki tej strukturze możemy używać tych jednostek pola powierzchni, które w danym przypadku są nam potrzebne:

```
PolePowierzchni poleKargula = new PolePowierzchni();  
poleKargula.Ar = 400;  
Console.WriteLine("Kargul posiada pole o powierzchni {0} ha.",  
    poleKargula.Hektar);
```

Enkapsulacja definiowanie indeksatorów i posługiwanie się wzorcem proxy

Naszym zadaniem jest utworzenie klasy Student. W wymaganiach określono, że klasa student zawiera takie atrybuty jak imię, nazwisko i numer indeksu. Po analizie stwierdzono, że imię i nazwisko będzie przechowywane w postaci napisu (typ string), natomiast numer indeksu jako liczba int. Dodatkowo wiemy, że numer indeksu nie może zostać zmieniony.

Utworzenie klasy reprezentującej studenta

Przez cały swój pobyt na uczelni student jest identyfikowany przez ten sam, raz określony numer indeksu, dlatego reprezentujące go pole zaimplementujemy jako tylko do odczytu. Pełna implementacja klasy powinna wyglądać następująco:

```
public class Student {  
    public string Imie { get; set; }  
    public string Nazwisko { get; set; }  
    readonly private int numerIndeksu;  
    public int NumerIndeksu {  
        get { return numerIndeksu; }  
    }  
    public Student(string imie, string nazwisko, int numerIndeksu) {  
        Imie = imie;  
        Nazwisko = nazwisko;  
        this.numerIndeksu = numerIndeksu;  
    }  
}
```

Może wydawać się niepotrzebne implementacja imienia i nazwiska jako właściwości, ale czy jesteśmy pewni, że w przyszłości nie będziemy chcieli kontrolować dostęp do tego pola? Może będziemy musieli sprawdzić, czy imię lub nazwisko nie zawiera niedozwolonych znaków np. gwiazdki czy znaku procenta lub czy zaczyna się wielką literą. Obecnie nie ma dodatkowych wymagań dotyczących tych pól, dlatego zaimplementowano je jako właściwości automatyczne.

W konstruktorze korzystamy bezpośrednio z pola `numerIndeksu`, ponieważ właściwość `NumerIndeksu` jest tylko do odczytu.

Utworzenie klasy pełnomocnika

Klasa `Student` będzie wykorzystywana w aplikacji konsolowej, dlatego należy utworzyć pełnomocnika, który umożliwi współpracę naszej klasy z konsolą. Implementacja tej klasy powinna wyglądać następująco:

```
public class StudentDlaKonsoli {
    private Student student;
    public StudentDlaKonsoli () {
        Console.Write("Podaj nazwisko: ");
        string nazwisko = Console.ReadLine();
        Console.Write("Podaj imię: ");
        string imie = Console.ReadLine();
        Console.Write("Podaj numer indeksu: ");
        int numer = int.Parse(Console.ReadLine());
        student = new Student(imie, nazwisko, numer);
    }
    public StudentDlaKonsoli(Student s) {
        if(s == null)
            throw new ArgumentException("Argument nie może być null");
        student = s;
    }
    public Student Student {
        get { return student; }
    }
    public void ZmienImie() {
        Console.Write("Podaj imię: ");
        student.Imie = Console.ReadLine();
    }
    public void ZmienNazwisko() {
        Console.Write("Podaj nazwisko: ");
        student.Nazwisko = Console.ReadLine();
    }
    public void WypiszStudenta () {
        Console.WriteLine("Pan(i) {0} {1} numer indeksu {2}",
            student.Imie, student.Nazwisko, student.NumerIndeksu);
    }
}
```

Dodanie konstruktora przyjmującego jako argument zmienną typu `Student` umożliwia współpracę istniejących już obiektów klasy `Student` z konsolą. Właściwość `Student` umożliwia bezpośredni dostęp do obiektu klasy `Student`.

Utworzenie programu testującego klasę `Student`

Kod testujący klasę `Student` może wyglądać następująco:

```
static void Main(string[] args) {
    StudentDlaKonsoli s1 = new StudentDlaKonsoli();
    s1.WypiszStudenta();
    s1.ZmienImie();
    s1.ZmienNazwisko();
    Student s2 = s1.Student;
    Console.WriteLine("Pan(i) {0} {1} numer indeksu {2}",
        s2.Imie, s2.Nazwisko, s2.NumerIndeksu);

    Student s3 = new Student("Jan", "Kowalski", 1234);
    StudentDlaKonsoli s4 = new StudentDlaKonsoli(s3);
    s4.WypiszStudenta();
}
```

```
    Console.ReadKey();  
}
```

Zmiana typu pola reprezentującego numer indeksu

Niestety, ponieważ zmieniono sposób tworzenia numerów indeksów (do numeru dodano literę oznaczającą wydział), musimy zmienić typ pola reprezentującego ten atrybut na `string`. Nie jesteśmy jednak w stanie zmienić wszystkich programów korzystających z naszej klasy `Student`, dlatego zdecydujemy się pozostawić konstruktor przyjmujący liczbę całkowitą jako numer indeksu i nie zmieniać typu właściwości `NumerIndeksu`. Zmodyfikujemy tylko implementację tych składowych oraz dodamy kilka nowych:

```
    readonly private string numerIndeksu;  
    public int NumerIndeksu {  
        get {  
            if(char.IsDigit(numerIndeksu[numerIndeksu.Length-1])) {  
                return int.Parse(numerIndeksu);  
            }  
            return int.Parse( numerIndeksu.Substring(0, numerIndeksu.Length -1));  
        }  
    }  
    public string NumerIndeksu2 {  
        get { return numerIndeksu; }  
    }  
    public Student(string imie, string nazwisko, int numerIndeksu) :  
        this(imie, nazwisko, numerIndeksu.ToString()) { }  
    public Student(string imie, string nazwisko, string numerIndeksu) {  
        Imie = imie;  
        Nazwisko = nazwisko;  
        this.numerIndeksu = numerIndeksu;  
    }  
}
```

Zauważmy, że modyfikacja naszej klasy nie wpłynęła na program z niej korzystający. Co więcej, jeżeli klasa zostanie zaimplementowana w osobnej bibliotece, to podmiana pliku DLL nie powinna wpłynąć na działanie programów z niego korzystających.

Utworzenie klasy reprezentującej listę studentów

Głównym naszym celem jest umożliwienie odwoływania się do studenta znajdującego się na liście przy pomocy jego numeru indeksu. Listę studentów będziemy przechowywać w tablicy. Na początku zdefiniujemy klasę `ListaStudentow`:

```
public class ListaStudentow {  
    private string Nazwa { set; get; }  
    private Student[] listaStudentow;  
    public ListaStudentow(string nazwa, params Student[] lista) {  
        Nazwa = nazwa;  
        listaStudentow = lista;  
    }  
}
```

Dodajmy następnie do klasy `ListaStudentow` dwa indeksatory. Pierwszy będzie przyjmował jako argument liczbę `int`, reprezentującą numer studenta na liście (oczywiście pomniejszony o jeden), a drugi – łańcuch znaków, reprezentujący numer indeksu.

```
    public Student this[int n] {  
        get { return listaStudentow[n]; }  
        set { listaStudentow[n] = value; }  
    }  
    public Student this[string nrIndeksu] {  
        get {
```

```

        for (int i = 0; i < listaStudentow.Length; i++) {
            if (nrIndeksu == listaStudentow[i].NumerIndeksu2) {
                return listaStudentow[i];
            }
        }
        return null;
    }
    set {
        for (int i = 0; i < listaStudentow.Length; i++) {
            if (nrIndeksu == listaStudentow[i].NumerIndeksu2) {
                listaStudentow[i] = value;
                return;
            }
        }
        throw new ArgumentOutOfRangeException(
            "Nie ma studenta o podanym indeksie na liście");
    }
}

```

Spróbujmy teraz skorzystać z klasy ListaStudentow:

```

static void Main(string[] args) {
    ListaStudentow lista1 = new ListaStudentow("Lista stypendystów",
        new Student("Jan", "Kowalski", "1111"),
        new Student("Jacek", "Wiśniewski", "2222"),
        new Student("Anna", "Nowak", "3333"));
    Console.Write("Podaj numer indeksu: ");
    string nrIndeksu = Console.ReadLine();
    Student s1 = lista1[nrIndeksu]; //wywołanie metody get
    if (s1 == null) {
        Console.WriteLine(
            "Student o podanym numerze indeksu nie jest stypendystą");
    }
    else {
        (new StudentDlaKonsoli(s1)).WypiszStudenta();
        Console.WriteLine(" jest stypendystą");
    }
    Student s2 = new Student("Joanna", "Maj", "4444");
    lista1["2222"] = s2; //wywołanie metody set
    (new StudentDlaKonsoli(lista1[1])).WypiszStudenta();
    Console.ReadKey();
}

```

Gotowe rozwiązanie powyższych przykładów znajduje się w katalogu **Demo\Modul03**.

Porady praktyczne

- Dla prywatnego pola i publicznej właściwości uzyskującej dostęp do niego używaj jako nazwy tego samego wyrazu, z tym że w przypadku pola stosuj notację CamelCase, natomiast w przypadku właściwości – notację PascalCase.
- Pole oraz właściwość skojarzoną z tym polem definiuj wewnątrz klasy obok siebie. Przeglądając później definicję klasy łatwiej zrozumieć, do czego służą poszczególne składowe, gdy elementy ściśle ze sobą powiązane mieszczą się na pojedynczym ekranie.
- Pojęcia, które reprezentują cechy obiektu (lub klasy), ale w celu obliczenia wartości tej cechy, konieczne jest wykonanie pewnych operacji, implementuj zawsze przy pomocy właściwości, a nie metod. Korzystanie z takiej klasy będzie bardziej intuicyjne.
- W przypadku gdy wartość jakiegoś atrybutu można obliczyć na podstawie wartości innych pól, nie definiuj dla niego osobnego pola. Zamiast tego zdefiniuj odpowiednią właściwość – korzystanie z klasy i jej pielęgnacja będą dużo łatwiejsze.

- Pamiętaj, że metoda `set` przyjmuje niejawnie argument o nazwie `value`, którego typ jest zgodny z typem właściwości lub indeksatora. Metoda `get` natomiast przekazuje do programu wartość, której typ jest zgodny z typem właściwości lub indeksatora.
- Stosuj zawsze enkapsulację wszystkich pól w Twojej klasie. Dzięki temu zmiana typu pola w przyszłości będzie mogła być przeprowadzona w prostszy sposób.
- W przypadku, gdy przy dostępie do pola nie ma konieczności wykonania dodatkowych operacji, stosuj właściwości automatyczne.
- Pamiętaj, że w przypadku właściwości automatycznych musisz zawsze zdefiniować obie metody: `set` i `get`. Właściwość automatyczna tylko do odczytu lub tylko do zapisu jest niedozwolona.
- W metodach danej klasy do pól odwołuj się przy pomocy właściwości skojarzonej z danym polem. Dzięki temu w przyszłości modyfikacja klasy będzie łatwiejsza. W razie konieczności zmiany sposobu dostępu do pola, kod będzie trzeba zmodyfikować tylko w jednym miejscu. Oczywiście odstępem od tej zasady mogą być względy wydajnościowe.
- Właściwość nie może być przekazywana do metody przez referencję i jako parametr wyjściowy.
- Indeksatory, podobnie jak inne metody, możesz przeciążyć.
- Indeksatory w dokumentacji MSDN są zazwyczaj wymieniane jako właściwość `Item`.
- Metody `set` i `get`, zarówno w indeksatorach jak i we właściwościach, mogą mieć różny poziom dostępu - różne modyfikatory dostępu.
- Pola, których wartości nie wolno zmienić przez całe życie obiektu, opatrz modyfikatorem `readonly`. W przypadku klasy opisującej osobę dobrymi kandydatami są pola reprezentujące rok urodzenia, pesel czy NIP.
- Pole `readonly` w odróżnieniu od pola `const`, może mieć różną wartość dla różnych obiektów. Pole `const` ma jedną wartość, nadaną podczas definicji pola.
- Wartość polu tylko do odczytu możesz nadać w konstruktorze.
- Staraj się unikać korzystania z konkretnej biblioteki wejścia-wyjścia. Twoja klasa może być w przyszłości wykorzystywana w aplikacjach różnego typu, niekoniecznie w aplikacjach konsolowych.
- Interakcję z użytkownikiem powinienś przenieść do oddzielnej klasy. Klasa proxy w celu zmiany i wyświetlenia stanu obiektu powinna korzystać z publicznych właściwości. Użytkownik i tak jest najwolniejszym ogniwem, więc nie powinno mieć to wpływu na wydajność aplikacji.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- rozumiesz pojęcie enkapsulacji
- wiesz do czego służą właściwości i kiedy je stosować
- potrafisz definiować własne właściwości
- wiesz kiedy stosować właściwości automatyczne
- wiesz do czego służą indeksatory
- potrafisz definiować własne indeksatory
- wiesz jaka jest różnica między metodą `set` i `get`
- rozumiesz do czego służą pola tylko do odczytu
- znasz różnice między polem `const` a polem `readonly`
- wiesz do czego służy wzorzec projektowy o nazwie proxy
- umiesz podać przykłady zastosowania wzorca projektowego proxy

Dodatkowe źródła informacji

1. Daniel Solis, *Illustrated C# 2008*, Apress, 2008

Książka dla tych wszystkich którzy pragną nauczyć się tworzyć programy w języku C#. Zawiera dokładne omówienie tego języka.

2. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka dla programistów chcących nauczyć się programować w języku C#.

3. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad jak powinniśmy pisać swoje programy.

4. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Wzorce projektowe Wydanie II*, WNT, 2008

Książka, która wprowadziła pojęcie wzorców projektowych do informatyki. Lektura obowiązkowa dla każdego, kto chce poznać temat wzorców projektowych.

5. Steven John Metsker, *C#. Wzorce projektowe*, Helion, 2005

Książka jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia jak wykorzystać cechy języka C# do tworzenia poprawnego kodu poprzez zastosowanie wzorców.

6. Judith Bishop, *C# 3.0 Design Patterns*, O'Reilly Media, Inc., 2008

Książka, podobnie jak poprzednia, jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia również nowe cechy języka C#.

7. *Codeguru*, <http://www.codeguru.pl/>



Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.

Laboratorium podstawowe

Problem 1 (czas realizacji 35 minut)

Twoja firma opracowuje program kadrowy dla pewnej firmy. Twoim zadaniem jest dokończenie i przetestowanie klasy *Osoba*. Do klasy *Osoba* oraz *Adres* musisz dodać właściwości przy pomocy których będzie można uzyskać dostęp do pól klasy. Zwróć uwagę, że wartość polu *numerEwidencyjny* klasy *Osoba* jest nadawana w momencie tworzenia obiektu i później ta wartość nie może być zmieniona. Zabezpiecz, aby nie można było utworzyć osoby, która się jeszcze nie urodziła. Oprócz roku urodzenia również w programie kadrowym potrzebny będzie wiek osoby. Dodaj również klasy, które będą pośredniczyć w interakcji między klasami *Osoba* i *Adres* a konsolą.

Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition	<ul style="list-style-type: none"> Otwórz Visual C# 2008 Express Edition. Z menu wybierz File -> New Project. Z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz Kadry. Kliknij OK. Z menu wybierz File -> Save Kadry. W polu Location wybierz folder w którym będzie zapisany projekt. Zaznacz pole wyboru Create directory for solution. W polu Solution Name wpisz Modul03. Naciśnij przycisk Save.
2. Do projektu Kadry dodaj istniejący plik z kodem	<ul style="list-style-type: none"> Z menu wybierz Project -> Add Existing Item. W oknie dialogowym Add ExistingItem – Kadry wybierz plik Osoba.cs, który znajduje się w katalogu Kurs\Lab\Start\Modul03, gdzie Kurs jest katalogiem gdzie zainstalowano pliki kursu. Kliknij Add.
3. Do klasy <i>Adres</i> dodaj odpowiednie właściwości	<ul style="list-style-type: none"> Przejdź do pliku Osoba.cs. Zmień wszystkie prywatne pola klasy Adres na publiczne właściwości automatyczne. Nie zapomnij, że według konwencji przyjętej w języku C# do publicznych składowych stosuje się notację PascalCase. Zwróć uwagę, że jeżeli zmienisz nazwy (małą literę na wielką), będziesz również musiał zmodyfikować kod konstruktorów klasy Adres. Po dodaniu kodu i wprowadzeniu odpowiednich zmian, klasa Adres powinna wyglądać następująco: <pre> Adres { public string Miejscowosc { set; get; } public string Kod { set; get; } public string NazwaUlicy { set; get; } public int NumerDomu { set; get; } public int? NumerMieszkania { set; get; } public Adres(int numerDomu, int? numerMieszkania, string nazwaUlicy, string kod, string miejscowosc) { this.NumerDomu = numerDomu; this.NumerMieszkania = numerMieszkania; this.NazwaUlicy = nazwaUlicy; this.Kod = kod; this.Miejscowosc = miejscowosc; } </pre>

	<pre> public Adres(int numerDomu, int? numerMieszkania, string nazwaUlicy) :this(numerDomu,numerMieszkania, nazwaUlicy,"02-222", "Warszawa") { } public Adres(int numerDomu, int? numerMieszkania) : this(numerDomu, numerMieszkania, "Aleje Jerozolimskie") { } public Adres(Adres adres) : this(adres.NumerDomu, adres.NumerMieszkania, adres.NazwaUlicy, adres.Kod, adres.Miejscowosc) { } } </pre> <p> Zmiany nazwy możesz dokonać w łatwy sposób, korzystając z tzw. <i>refaktoringu</i>. Jeżeli chcesz zmienić dowolną nazwę, kliknij ją prawym klawiszem myszy, a następnie z menu kontekstowego wybierz Refaktor -> Rename.</p>
4. Zmodyfikuj kod klasy <i>Osoba</i>	<ul style="list-style-type: none"> • Zmień pole numerEwidencyjny tak, aby było tylko do odczytu i dodaj właściwość, przy pomocy której uzyskamy dostęp do tego pola: <pre> readonly private int numerEwidencyjny; public int NumerEwidencyjny { get { return numerEwidencyjny; } } </pre> <p> Czy możemy utworzyć właściwość do odczytu i zapisu? Czy potrafisz uzasadnić odpowiedź?</p> • Dodaj właściwość, przy pomocy której uzyskamy dostęp do pola rokUrodzenia. Zapewnij, że podany rok jest wcześniejszy od bieżącego: <pre> private int rokUrodzenia; public int RokUrodzenia { get { return rokUrodzenia; } set { if(value > DateTime.Now.Year) throw new ArgumentOutOfRangeException("Rok urodzenia musi być wcześniejszy od bieżącego"); rokUrodzenia = value; } } </pre> • Dodaj właściwość tylko do odczytu reprezentującą wiek: <pre> public int Wiek { get { return DateTime.Now.Year - rokUrodzenia; } } </pre> • Zmień pozostałe prywatne pola klasy Osoba na publiczne właściwości automatyczne. Zmodyfikowany kod powinien wyglądać następująco: <pre> public string Nazwisko { set; get; } public string Imie { set; get; } public Adres AdresZamieszkania { set; get; } public Osoba(int numerEwidencyjny, int rokUrodzenia, string imie, string nazwisko, Adres adres) { this.RokUrodzenia = rokUrodzenia; this.numerEwidencyjny = numerEwidencyjny; this.Imie = imie; this.Nazwisko = nazwisko; this.AdresZamieszkania = adres; } </pre>

	<pre> } public Osoba(int numerEwidencyjny, int rokUrodzenia, string imie, string nazwisko, int numerDomu, int? numerMieszkania, string nazwaUlicy, string kod, string miejscowosc) : this(numerEwidencyjny, rokUrodzenia, imie, nazwisko, new Adres(numerDomu,numerMieszkania,nazwaUlicy, kod,miejscowosc)) { } public Osoba(int numerEwidencyjny, int rokUrodzenia, string imie,string nazwisko, Adres adres) { this.RokUrodzenia = rokUrodzenia; this.numerEwidencyjny = numerEwidencyjny; this.Imie = imie; this.Nazwisko = nazwisko; this.AdresZamieszkania = adres; } public Osoba(int numerEwidencyjny, int rokUrodzenia, string imie, int numerDomu) : this(numerEwidencyjny, rokUrodzenia, imie, "Kowalski", numerDomu, null, "Aleje Jerrozolimskie", "02-222", "Warszawa") {} public Osoba(Osoba osoba) { numerEwidencyjny = osoba.numerEwidencyjny; Imie = osoba.Imie; Nazwisko = osoba.Nazwisko; AdresZamieszkania = new Adres(osoba.AdresZamieszkania); } </pre>
<p>5. Dodaj klasę pośredniczącą w interakcjach między klasą Adres a konsolą</p>	<ul style="list-style-type: none"> • Z menu wybierz Project -> Add Class. • W oknie dialogowym Add New Item – Kadry z listy Visual Studio installed templates wybierz Class. • W polu Name wpisz AdresDlaKonsoli. • Kliknij OK. • Do klasy AdresDlaKonsoli dodaj prywatne pole typu Adres: <pre> private Adres adres; </pre> • Dodaj właściwość, przy pomocy której uzyskamy dostęp do prywatnego pola adres. <pre> public Adres Adres { get { return adres; } set { adres = value; } } </pre> • Do klasy AdresDlaKonsoli dodaj dwa publiczne konstruktory – jeden bezparametrowy, drugi przyjmujący jako argument zmienną typu Adres: <pre> public AdresDlaKonsoli() { Console.Write("Podaj nazwę miejscowości: "); string miejscowosc = Console.ReadLine(); Console.Write("Podaj kod: "); string kod = Console.ReadLine(); Console.Write("Podaj nazwę ulicy: "); string nazwaUlicy = Console.ReadLine(); int numerDomu; do { Console.Write("Podaj numer domu: "); } while (!int.TryParse(Console.ReadLine(), out numerDomu)); Console.Write("Czy jest numer mieszkania <t/n>: "); </pre>

```
char c = Console.ReadKey().KeyChar;
int? numerMieszkania;
if (c == 't') {
    int i;
    Console.WriteLine();
    do {
        Console.Write("Podaj numer mieszkania: ");
    }
    while (!int.TryParse(Console.ReadLine(), out i));
    numerMieszkania = i;
}
else {
    numerMieszkania = null;
}
this.Adres = new Adres(numerDomu,numerMieszkania,nazwaUlicy,
                      kod,miejscowosc);
}
```

- Dodaj publiczne metody, przy pomocy których będziesz mógł zmienić pojedyncze atrybuty klasy **Adres**:

```
public void ZmienMiejscowosc() {
    Console.Write("Podaj nazwę miejscowości: ");
    Adres.Miejscowosc = Console.ReadLine();
}

public void ZmienKod() {
    Console.Write("Podaj kod: ");
    Adres.Kod = Console.ReadLine();
}

public void ZmienUlice() {
    Console.Write("Podaj nazwę ulicy: ");
    Adres.NazwaUlicy = Console.ReadLine();
}

public void ZmienNumerDomu() {
    int numerDomu;
    do {
        Console.Write("Podaj numer domu: ");
    }
    while (!int.TryParse(Console.ReadLine(),out numerDomu));
    Adres.NumerDomu = numerDomu;
}

public void ZmienNrMieszkania() {
    Console.Write("Czy jest numer mieszkania <t/n>: ");
    char c = Console.ReadKey().KeyChar;
    if (c == 't') {
        int i;
        Console.WriteLine();
        do {
            Console.Write("Podaj numer mieszkania: ");
        }
        while (!int.TryParse(Console.ReadLine(), out i));
        Adres.NumerMieszkania = i;
    }
    else {
        Adres.NumerMieszkania = null;
    }
}
```

	<ul style="list-style-type: none"> • Dodaj metodę która zmienia wszystkie pola klasy Adres: <pre>public void ZmienAdres() { ZmienMiejscowosc(); ZmienKod(); ZmienUlice(); ZmienNumerDomu(); ZmienNrMieszkania(); }</pre> • Dodaj metodę która wypisuje wszystkie wartości pól klasy Adres: <pre>public void WypiszAdres() { Console.WriteLine("{0} {1} ul. {2} nr {3}", Adres.Kod, Adres.Miejscowosc, Adres.NazwaUlicy, Adres.NumerDomu); if (Adres.NumerMieszkania != null) { Console.WriteLine("/{0}", Adres.NumerMieszkania); } }</pre>
6. Dodaj klasę pośredniczącą w interakcjach między klasą Osoba a konsolą	<ul style="list-style-type: none"> • Z menu wybierz Project -> Add Class. • W oknie dialogowym Add New Item – Kadry z listy Visual Studio installed templates wybierz Class. • W polu Name wpisz OsobaDlaKonsoli. • Kliknij OK. • Do klasy OsobaDlaKonsoli dodaj prywatne pole typ Osoba: <pre>private Osoba osoba;</pre> • Dodaj właściwość przy pomocy której uzyskujemy dostęp do prywatnego pola osoba: <pre>public Osoba Osoba { set { osoba = value; } get { return osoba; } }</pre> • Do klasy OsobaDlaKonsoli dodaj dwa publiczne konstruktory – jeden bezargumentowy, drugi przyjmujący jako argument zmienną typu Osoba: <pre>public OsobaDlaKonsoli() { Console.WriteLine("Podaj imię: "); string imie = Console.ReadLine(); Console.WriteLine("Podaj nazwisko: "); string nazwisko = Console.ReadLine(); int numerEwidencyjny; do { Console.WriteLine("Podaj numer ewidencyjny: "); } while (!int.TryParse(Console.ReadLine(), out numerEwidencyjny)); int rokUrodzenia; do { Console.WriteLine("Podaj rok urodzenia: "); } while (!int.TryParse(Console.ReadLine(), out rokUrodzenia)); Console.WriteLine("Podaj adres zamieszkania:"); AdresDlaKonsoli adr = new AdresDlaKonsoli(); this.Osoba = new Osoba(numerEwidencyjny, rokUrodzenia, imie, nazwisko, adr.Adres); } public OsobaDlaKonsoli(Osoba osoba) { this.Osoba = osoba; }</pre> • Dodaj publiczne metody, przy pomocy których będziesz mógł zmienić


	<p>atrybuty nazwisko oraz adres zamieszkania klasy Osoba:</p> <pre>public void ZmienNazwisko() { Console.Write("Podaj nazwisko: "); Osoba.Nazwisko = Console.ReadLine(); } public void ZmienAdres() { Console.WriteLine("Podaj adres zamieszkania:"); AdresDlaKonsoli adr = new AdresDlaKonsoli(); Osoba.AdresZamieszkania = adr.Adres; }</pre> <ul style="list-style-type: none"> • Dodaj metodę, która wypisuje wszystkie wartości pól klasy Osoba: <pre>public void WypiszOsobe() { Console.Write("Pan(i) {0} {1} numer ewidencyjny {2} lat {3} ", Osoba.Imie, Osoba.Nazwisko, Osoba.NumerEwidencyjny, Osoba.Wiek); AdresDlaKonsoli adr = new AdresDlaKonsoli(Osoba.AdresZamieszkania); adr.WypiszAdres(); }</pre>
7. Przetestuj klasę Osoba	<ul style="list-style-type: none"> • Przejdź do pliku Program.cs. • Do metody Main dodaj kod, który przetestuje, czy klasa Osoba działa prawidłowo. Przykładowy kod jest zamieszczony poniżej: <pre>OsobaDlaKonsoli os1 = new OsobaDlaKonsoli(); Console.WriteLine(); os1.ZmienAdres(); Console.WriteLine(); os1.ZmienNazwisko(); Console.WriteLine("\n*****\n"); os1.WypiszOsobe(); Console.WriteLine("\n*****\n"); Osoba osoba2 = new Osoba(123, 1990, "Anna", "Kowalska", 12, 23, "Kwiatowa", "97-350", "Piotrków Tryb."); os1.Osoba = osoba2; os1.ZmienAdres(); Console.WriteLine(); os1.ZmienNazwisko(); Console.WriteLine("\n*****\n"); os1.WypiszOsobe(); Console.ReadKey();</pre>
8. Skompiluj i uruchom program	<ul style="list-style-type: none"> • Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. • W celu uruchomienia programu, z menu Debug wybierz Start Debugging.


Problem 2 (czas realizacji 10 minut)

Masz napisaną klasę **Lista**. Klasa ta implementuje listę jednokierunkową. Twoim zadaniem jest dodanie możliwości odwołania się do elementu listy przy pomocy indeksu, podobnie jak odwołujemy się do elementów tablicy. W przypadku, gdy element o podanym indeksie nie istnieje, rzuć wyjątek **IndexOutOfRangeException**. Lista jednokierunkowa jako struktura danych została omówiona w kursie „Wprowadzenie do programowania”.

Pliki wymagane w tym laboratorium znajdują się w katalogu **Kurs\Lab\Start\Modul03**, gdzie **Kurs** jest katalogiem, w którym zainstalowano pliki kursu.

Zadanie	Tok postępowania
1. Do bieżącego	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element

rozwiązania dodaj nowy projekt	<p>reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project.</p> <ul style="list-style-type: none"> W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestListy. Kliknij OK.
2. Zaznacz projekt TestKolejki jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestListy, a następnie z menu kontekstowego wybierz Set as StartUp Project.
3. Do projektu TestListy dodaj istniejący plik z kodem	<ul style="list-style-type: none"> Z menu wybierz Project -> Add Existing Item.  Zwróć uwagę, aby przed wykonaniem powyższego polecenia w okienku Solution Explore był zaznaczony projekt TestListy. W oknie dialogowym Add ExistingItem – TestListy wybierz plik Listy.cs, który znajduje się w katalogu Kurs\Lab\Start\Modul03, gdzie Kurs jest katalogiem w którym zainstalowano pliki kursu. Kliknij Add.
4. Dodaj do klasy List możliwość odwoływania się do elementu listy przy pomocy indeksu	<ul style="list-style-type: none"> Przejdź do pliku Listy.cs. Do klasy Lista dodaj prywatną metodę, która zwróci węzeł o podanym indeksie. W przypadku gdy węzeł o podanym indeksie nie istnieje, zgłoś wyjątek IndexOutOfRangeException: <pre>private Wezel ZnajdzWezel(int indeks) { int i = 0; Wezel tmp = glowa; while (tmp != null && i < indeks) { tmp = tmp.Nastepny; i++; } if(tmp == null) { throw new IndexOutOfRangeException("Nie ma elementu o podanym indeksie"); } return tmp; }</pre> Do klasy Lista dodaj publiczny indeksator. Do jego implementacji wykorzystaj zdefiniowaną poprzednio metodę ZnajdzWezel: <pre>public string this[int indeks] { get { return ZnajdzWezel(indeks).Dane; } set { ZnajdzWezel(indeks).Dane = value; } }</pre>
5. Przetestuj działanie indeksatora	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Do metody Main dodaj kod, który przetestuje działanie indeksatora: <pre>Listy.Lista imiona = new Listy.Lista(); imiona.DodajDoGlowy("Ania"); imiona.DodajDoGlowy("Agnieszka"); imiona.DodajDoGlowy("Wiktoria"); imiona.DodajDoGlowy("Kasia"); for(int i = 0; i < imiona.PobierzLiczbeElementow(); i++) { Console.WriteLine("{0}. {1}", i, imiona[i]); } imiona[1] = "Basia"; imiona[3] = "Ola"; for (int i = 0; i < imiona.PobierzLiczbeElementow(); i++) { Console.WriteLine("{0}. {1}", i, imiona[i]); }</pre>

	<pre>} Console.ReadKey();</pre> <p> Zwróć uwagę, że powyższy sposób przechodzenia po liście i wypisywanie jej elementów jest bardzo niewydajny. Przykład ten tylko pokazuje użycie indeksatora. Poprawny sposób przechodzenia po liście zostanie przedstawiony w module 11 przy omawianiu interfejsu IEnumerable.</p>
6. Skompiluj i uruchom program	<ul style="list-style-type: none">• Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program.• W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 90 min)

Firma w której pracujesz dostała zlecenie na napisanie programu wspierającego sprzedaż. Twoim zadaniem jest stworzenie klasy reprezentującej fakturę. Faktura zawiera spis towarów. Każdy towar jest opisany przy pomocy następujących atrybutów:

- nazwa towaru
- jednostka, w której sprzedawany jest dany towar (tony, sztuki, metry itp.)
- cena jednostkowa netto
- cena jednostkowa brutto
- ilość danych jednostek (tony, sztuki, metry itp.)
- cena całkowita netto
- cena całkowita brutto
- procent, jaki w cenie brutto stanowi podatek

Dodatkowo każda faktura zawiera następujące informacje:

- data sprzedaży
- termin zapłaty wyrażony w liczbie dni
- termin zapłaty wyrażony przez podanie daty
- cena całkowita netto
- cena całkowita brutto
- numer faktury, którego nie można zmienić

Musisz również umożliwić odwołanie się do każdego towaru znajdującego się na liście przy pomocy indeksu lub jego nazwy, z tym że przy pomocy nazwy ma być możliwe jedynie pobieranie informacji o towarze (bez możliwości modyfikacji).

Firma na razie zdecydował się utworzyć program jako program konsolowy. W przyszłości program ma jednak być programem okienkowym.

Zadanie 2 (czas realizacji 45 min)

Twoim zadaniem jest stworzenie biblioteki, który ułatwi posługiwanie się różnymi jednostkami dla tych samych pojęć. Biblioteka ta będzie później używana różnych projektach. Powinieneś uwzględnić następujące pojęcia i ich jednostki:

- ilość
 - kopa – 60
 - tuzin – 12
 - mendel – 15
- długość
 - metr
 - milimetr – 0,001 metra
 - cal – 25,4 milimetra
 - jard – 0,9144 metra
 - mila morska – 1 852 metrów
 - mila angielska – 1609,344 metra
- masa
 - kilogram
 - gram – 0,001 kilograma

- tona – 1000 kilogramów
- kwintal – 100 kilogramów
- funt brytyjski – 0,453592
- uncja – 28,35 grama
- temperatura
 - Kelvin
 - stopnie Celsjusza – $-273,15 + \text{liczba stopni w skali Kelvina}$
 - stopnie Fahrenheita – $32 + 9/5 \cdot \text{liczba stopni w skali Celsjusza}$
- moc
 - wat
 - koń mechaniczny – 735,498 watów
- objętość
 - metr sześcienny
 - litr – 0,001 metra sześciennego
 - kwaterka – 0,25 litra
 - galon angielski – 4,5456 litra

Utwórz program przeliczający jednostki z wykorzystaniem przygotowanej biblioteki.

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 4

Wersja 2

Składowe statyczne

Spis treści

Składowe statyczne.....	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	7
Porady praktyczne	10
Uwagi dla studenta	11
Dodatkowe źródła informacji	11
Laboratorium podstawowe	12
Problem 1 (czas realizacji 45 minut)	12
Laboratorium rozszerzone	18
Zadanie 1 (czas realizacji 90 min)	18

Informacje o module

Opis modułu

W tym module zapoznasz się z pojęciem składowych statycznych. Dowiesz się, do czego służą i jak z nich korzystać w języku C#. Nauczysz się, co to jest konstruktor statyczny oraz po co go się definiuje. Dowiesz się również, co to są klasy statyczne oraz co to są metody rozszerzające i jak je implementować. Zostanie również przedstawiony wzorec projektowy o nazwie Singleton.

Cel modułu

Celem modułu jest przedstawienie pojęcia składowych statycznych w programowaniu obiektowym i pokazanie możliwości ich wykorzystania w języku C#.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- znał pojęcie pól, metod oraz składowych statycznych i potrafił z nich korzystać
- wiedział, do czego służy konstruktor statyczny i potrafił go używać
- wiedział, co to są klasy statyczne i potrafił je definiować
- wiedział, co to są metody rozszerzające i potrafił je definiować
- wiedział, co to jest i do czego służy wzorec projektowy Singleton

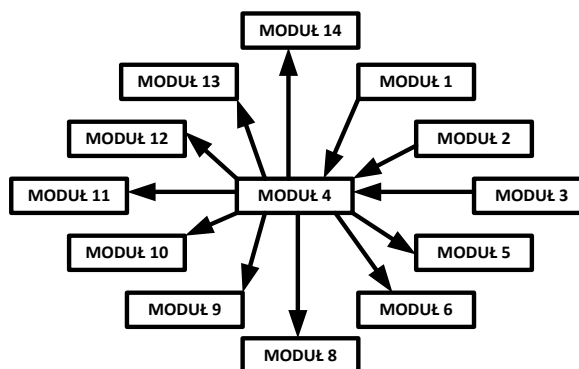
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinienś:

- potrafić definiować klasę
- wiedzieć, co to jest pole i co to jest metoda
- znać i rozumieć pojęcie konstruktora
- znać modyfikatory dostępu (public i private)
- rozumieć pojęcie wzorców projektowych

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w modułach „Pojęcie klasy”, „Konstruktor” oraz „Właściwości i indeksatory”.



Rys. 6 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Potrafisz definiować klasy i tworzyć obiekty. Każdy obiekt posiada zbiór własnych pól. Taka sytuacja wydaje korzystna. Rozważmy jednak sytuację kont bankowych. Jedną z cech konta bankowego jest oprocentowanie. W większości przypadków w danym banku konta jednego typu, inaczej należące do jednej klasy Konto, mają to samo oprocentowanie, nieważne, czy jest to konto Nowaka, czy Kowalskiego. Zastanówmy się, jak to oprogramować. Tworząc zwykłe pole reprezentujące oprocentowanie spowodujemy, że każdy obiekt klasy Konto zawierać będzie własną kopię informacji na temat oprocentowania. Zakładając, że pole reprezentujące oprocentowanie jest typu `double` i masz 1024 konta, na przechowanie informacji na temat oprocentowania potrzeba osiem kilobajtów, a tak naprawdę wystarczyłoby tylko osiem bajtów. Drugim problemem, poza nieefektywnym wykorzystaniem pamięci, jest spójność. Co pewien czas zmienia się oprocentowanie. Musielibyśmy zapewnić, że wartość oprocentowania zmieni się we wszystkich kontach. Sama zmiana zajęłaby zapewne więcej czasu, niż modyfikacja pojedynczej wartości. Podsumowując, istnieją sytuacje, gdzie wszystkie obiekty danej klasy mają wspólną i tę samą co do wartości cechę. Cecha ta tak naprawdę nie jest atrybutem obiektu tylko samej klasy. Jeśli chcesz się dowiedzieć, jak w języku C# definiować składowe, które opisują klasę, a nie obiekt, zapoznaj się z treścią tego modułu.

Podstawy teoretyczne

W języku C# można zdefiniować pole, które jest wspólne dla wszystkich obiektów danej klasy. Przy pomocy tego pola opisujemy cechę całej klasy, nie pojedynczego obiektu. Pola te nazywamy *polami statycznymi* i definiujemy przy pomocy słowa `static` w następujący sposób:

```
class nazwa_klasy {  
    ...  
    [modyfikator_dostępu] static typ_pola nazwa_pola  
                                [=wyrażenie_inicjalizujące];  
}
```

Mając zdefiniowane pole statyczne wewnątrz metody danej klasy możemy odwołać się do niego przy pomocy jego nazwy. Nazwy tej nie można jednak poprzedzić słowem `this`. W przypadku gdy nazwa zmiennej lokalnej przesłoni nazwę statycznego pola klasy, możemy zastosować następującą konstrukcję:

```
nazwa_klasy.nazwa_pola
```

czyli odwołać się do składowej przy pomocy nazwy klasy. Podobnie odwołujemy się do pól statycznych z metod innych klas. Zilustrujmy to przykładem:

```
class Klasa1 {  
    public static int Nazwa1;  
    private static int nazwa2;  
    public void Metoda1() {  
        Nazwa1 = 20;  
        int nazwa2;  
        nazwa2 = 30;           //odwołanie się do zmiennej lokalnej  
        Klasa1.nazwa2 = 30;    //odwołanie się do pola statycznego  
    }  
}  
  
class Klasa2 {  
    public void Metoda2() {  
        Klasa1.Nazwa1 = 20;    //odwołanie się do pola statycznego  
        //Klasa1.nazwa2 = 30;  //brak dostępu, pole prywatne  
    }  
}
```

```
    }  
}
```

Analizując powyższy przykład zauważymy, że możemy odwołać się do pola statycznego nie definiując wcześniej żadnego obiektu. Możemy również zabezpieczyć pole statyczne przed nieodpowiednim dostępem, czyniąc je prywatnym. Zachodzi pytanie, czy możemy manipulować tymi polami spoza klasy bez konieczności tworzenia obiektu. Domyślamy się, że odpowiedź brzmi tak. Służą do tego *metody statyczne*. Metody statyczne i pola statyczne nazywamy *składowymi* lub *składnikami statycznym*. Metodę statyczną, podobnie jak pole, definiujemy przy pomocy słowa kluczowego `static` w następujący sposób:

```
class nazwa_klasy {  
    ...  
    [modyfikator_dostępu] static typ_zwracany  
                                nazwa_metody([lista_parametrow])  
    {  
        //ciało metody  
    }  
}
```

Do metod statycznych odwołujemy się podobnie jak do pól statycznych, czyli z metod innej klasy metodę taką wywołujemy:

```
nazwa_klasy.nazwa_metody([lista_argumentow]);
```

Metody statyczne można wywołać przed utworzeniem obiektu, ale mają pewne ograniczenie. W ich ciele nie można odwołać się do zwykłych składowych (składowych niestatycznych). Następujący kod spowoduje błąd kompilacji:

```
class Klasa1 {  
    private int pole1;  
    public void Metoda1(){  
        //ciało metody  
    }  
    public static void Metoda2(){  
        pole1 = 10;    //Błąd, odwołanie się do zwykłego pola  
        Metoda1();    //Błąd, wywołanie zwykłej metody  
    }  
}
```

Powyższy błąd wynika z faktu, że metody statyczne nie są wywoływane na rzecz konkretnego obiektu. Nie jest przekazywany do nich niejawni argument `this`, a pamiętajmy, że odwołanie `pole1` jest traktowane przez kompilator jako `this.pole1`.

Zwróćmy uwagę, że przekazanie argumentu do funkcji wiąże się z koniecznością wykonania dodatkowych czynności. W związku z tym wywołanie zwykłej metody będzie zawsze wolniejsze, niż wywołanie identycznej metody opatrzonej słowem `static`. Podsumowując, jeżeli metoda nie odwołuje się bezpośrednio do żadnych zwykłych (niestatycznych) składowych, to powinniśmy rozważyć czy nie zdefiniować jej jako metody statycznej. Decyzja ta powinna również uwzględnić ewentualne przyszłe zmiany w implementacji danej metody oraz czy nie jest to metoda wirtualna. Metody wirtualne zostaną opisane w module 8.

Stwierdzenie, że z metod statycznych nie można odwołać się zwykłych pól, nie jest do końca prawdziwe, co ilustruje kolejny przykład:

```
class Klasa1 {  
    private int pole1;  
    public void Metoda1(){  
        //ciało metody  
    }  
    public static void Metoda2(Klasa1 x){
```



```
        x.pole1 = 10;  
        x.Metoda1();  
    }  
}
```

Oczywiście w podobny sposób możemy odwołać się do zwykłych składowych z metod innych klas. Metoda statyczna ma jedną przewagę nad metodami klas „obcych” – jako metoda zdefiniowana w danej klasie ma również dostęp do składowych prywatnych.

Konstruktor statyczny

Pola statyczne mogą być prywatne i użytkownicy naszej klasy – inni programiści – mogą się nimi posługiwać w bezpieczny sposób dla naszej klasy przy użyciu odpowiednio napisanych publicznych metod statycznych. Zachodzi jednak pytanie, co zrobić, aby nadać takiemu polu odpowiednią wartość. Oczywiście wartość tę możemy nadać w miejscu definicji takiego pola, np.:

```
private static int nazwaPola = 10;
```

Problem zaczyna się jednak, gdy do inicjalizacji takiego pola konieczne jest wykonanie kilku dodatkowych czynności, np. połączenie z bazą danych i pobranie z niej odpowiedniej wartości lub odczyt danej wartości z pliku. Rozwiązaniem tego problemu w języku C# jest *konstruktor statyczny*. Konstruktor statyczny, podobnie jak zwykły konstruktor, jest metodą klasy. Jest on wywołany w sposób niejawny w nie do końca ściśle określonym miejscu naszego programu, jednak zawsze przed odwołaniem się do jakiegokolwiek składowej statycznej danej klasy oraz przed utworzeniem obiektu danej klasy. Definiujemy go w następujący sposób:

```
static nazwa_klasy()  
{  
    //ciało konstruktora  
}
```

Definiując statyczny konstruktor nie wolno dodać żadnego modyfikatora poza słowem `static`, w tym modyfikatora dostępu. Konstruktor statyczny jest wywoływany niejawnie przez .NET Framework i nie mamy możliwości wywołania go w sposób jawny. Ponieważ konstruktor statyczny jest wywoływany tylko niejawnie, nie możemy dostarczyć do niego argumentów, czyli nie możemy go przeciążyć. Podobnie jak w każdej metodzie statycznej, w konstruktorze statycznym nie możemy odwołać się do zwykłych składowych.

Klasy statyczne

W wersji języka C# 2.0 wprowadzono pojęcie *klas stycznych*. Klasa statyczna jest to klasa, która może posiadać tylko składowe statyczne i definiujemy ją w następujący sposób:

```
[modyfikator_dostępu] static class nazwa_klasy_statycznej {  
    //definicja klasy  
}
```

W przypadku klas statycznych, nie możemy tworzyć obiektów tej klasy, a nawet deklarować zmiennych. Poniższa linijka spowoduje już błąd kompilacji:

```
KlasaStatyczna nazwaZmiennej;
```

Klasy statyczne powstały jako pewnego rodzaju substytut funkcji i zmiennych globalnych, gdyż w języku C# nie można bezpośrednio wewnątrz przestrzeni nazw definiować zmiennych ani funkcji. Klasy statyczne są niejako kontenerem na składowe statyczne, które nie są związane z żadnym obiektem. W bibliotece FCL (ang. *Framework Class Library*) klasycznym przykładem klasy statycznej jest klasa `System.Math`. Zawiera ona zbiór metod obliczających różne funkcje matematyczne (np. sinus, logarytm czy pierwiastek) oraz definicję stałych matematycznych (np. `pi` czy podstawa logarytmu naturalnego – `e`). Do obliczenia logarytmu nie potrzebujemy żadnego obiektu:

```
Console.WriteLine("ln(e) = {0}", Math.Log(Math.E));
```

Metody rozszerzające

Metody rozszerzające (ang. *extension methods*) zostały wprowadzone w wersji 3.0 języka C#. Umożliwiają one niejako dodanie do istniejącego typu nowej metody bez konieczności modyfikacji oryginalnego typu i powtórnej kompilacji. Definiujemy je w następujący sposób:

```
[modyfikator_dostępu] static class nazwa_klasy_statycznej {  
    [modyfikator_dostępu] static typ_zwracany  
        nazwa_metody(this nazwa_typu_rozszerzanego identyfikator,  
            [lista_parametrow])  
    {  
        //ciało metody  
    }  
}
```

W powyższej definicji widzimy, że metoda rozszerzająca jest metodą statyczną i może być definiowana tylko w klasach statycznych. Metoda ta ma specyficzny pierwszy parametr: nazwa jego typu, będąca jednocześnie nazwą typu rozszerzanego, musi być poprzedzona słowem `this`. Użycie metody przeanalizujemy na następującym przykładzie:

```
static NazwaKlasy {  
    public static void F(this double x) {  
        //ciało metody  
    }  
}
```

Powyższy kod rozszerza typ `double`. Powyższą metodę możemy wywołać jak każdą inną metodą statyczną:

```
NazwaKlasy.F(3.4);
```

Co wydaje się jednak ciekawsze, metodę `F` możemy wywołać również w ten sposób:

```
double x = 4.5;    //argument jest przesyłany przez wartość musi więc być  
x.F();             //wcześniej zainicjalizowany  
(3.4).F();
```

Warto zwrócić uwagę, że metody częściowe mają pewne ograniczenia, a mianowicie:

- Idea metod rozszerzających nie dotyczy pól oraz właściwości. Nie można w ten sposób również dodać zdarzenia do klasy. Zdarzenia zostaną omówione w module 11 tego kursu.
- Nie można przesłonić istniejącej metody.
- Metoda rozszerzająca nie ma dostępu do prywatnych składowych rozszerzanej klasy lub struktury.

Podsumowują, metody rozszerzające są pewną sztuczką, która umożliwia nam wywołanie w inny sposób metody statycznej pewnej klasy.

Wzorzec projektowy Singleton

Jednym z praktycznych zastosowań składowych statycznych jest ich użycie przy implementacji wzorca projektowego *Singleton*. Celem tego wzorca jest stworzenia klasy, która posiadałaby co najwyżej jeden swój obiekt oraz dostarczenie łatwego sposobu dostępu do tego obiektu. Wzorzec ten implementujemy przez utworzenie klasy, która posiada następujące składowe:

- Prywatny konstruktor, dzięki czemu nie można utworzyć obiektu tej klasy poza nią samą.
- Prywatne statyczne pole, które zawiera odwołanie do żądanego obiektu.
- Publiczną statyczną metodę, która zwraca żądany obiekt. Wywołanie jej jest jedynym sposobem pozyskania żądanego obiektu:

```
public class Singleton {  
    private static Singleton obiekt = new Singleton();  
    Singleton()  
}
```

```
    { }  
    public static Singleton ZwrocObiekt() {  
        return obiekt;  
    }  
}
```

Przykładowe rozwiązanie

Definicja metody rozszerzającej

W tym przykładzie rozszerzymy typ `double` przez dodanie do niego metody obliczającej potęgę całkowitą liczby rzeczywistej. Utwórzmy klasę statyczną i dodajmy do niej odpowiednią metodę:

```
static class RozszerzenieTypuDouble {  
    public static double PotegaCalkowita(this double x, int n) {  
        double iloczyn = 1;  
        for (int i = 0; i < (n > 0 ? n : -n); I++) {  
            iloczyn *= x;  
        }  
        if (n < 0)  
            return 1 / iloczyn;  
        return iloczyn;  
    }  
}
```

Przetestujmy teraz dodaną metodę:

```
double x = 10;  
Console.WriteLine("{0} do potęgi drugiej: {1}", x, x.PotegaCalkowita(2));  
Console.WriteLine("{0} do potęgi minus drugiej: {1}",  
    10, (10.0).PotegaCalkowita(-2));
```

Wykorzystanie konstruktora statycznego i metod rozszerzających

Załóżmy, że mamy zdefiniowaną następującą klasę:

```
public class Student {  
    public string Imie { get; set; }  
    public string Nazwisko { get; set; }  
    readonly private int numerIndeksu;  
    public int NumerIndeksu {  
        get { return numerIndeksu; }  
    }  
    public Student(string imie, string nazwisko, int numerIndeksu) {  
        Imie = imie;  
        Nazwisko = nazwisko;  
        this.numerIndeksu = numerIndeksu;  
    }  
}
```

Mamy następujące zadania do wykonania:

- zapewnić, że każdy student będzie miał nadany unikalny numer indeksu
- umożliwić inicjalizację, nadanie wartości poszczególnym polom i wyświetlenie wartości poszczególnych pól klasy `Student` przy pomocy konsoli. Tym razem zaimplementujemy to przy pomocy metod rozszerzających.

Zapewnienie unikalności numerowi indeksu

Często najłatwiejszym ogniwem w systemie komputerowym jest człowiek, musimy więc uniemożliwić programiście nadawanie numerów indeksów samodzielnie. Numery indeksów będą generowane automatycznie, dlatego z listy argumentów konstruktora usuniemy parametr o nazwie `numerIndeksu`:

```
public Student(string imie, string nazwisko) {  
    Imie = imie;  
    Nazwisko = nazwisko;  
}
```

Do klasy `Student` dodamy pole, które będzie przechowywać numer indeksu ostatnio dodanego studenta. Pole to będzie polem statycznym oraz prywatnym, aby uniemożliwić programiście zmianę jego wartości:

```
private static int ostatniNumerIndeksu;
```

Do klasy `Student` dodamy statyczny konstruktor, który będzie inicjalizował pole `ostatniNumerIndeksu`. Inicjalizacja będzie polegała na wczytaniu wartości ostatniego numeru indeksu z pliku. W przypadku, gdy dany plik nie istnieje, polu `ostatniNumerIndeksu` zostanie nadana wartość zero. Nazwę pliku, gdzie będzie zapisywany ostatnio nadany numer indeksu, przechowamy w polu `const`, tak aby łatwo było w przyszłości zmienić lokalizację pliku:

```
const string NAZWAPLIKU = "numer.dat";  
  
static Student() {  
    if(File.Exists(NAZWAPLIKU)) {  
        BinaryReader br = null;  
        try {  
            br = new BinaryReader(File.Open(NAZWAPLIKU, FileMode.Open));  
            ostatniNumerIndeksu = br.ReadInt32();  
        }  
        finally {  
            if(br != null)  
                br.Close();  
        }  
    }  
    else {  
        ostatniNumerIndeksu = 0;  
    }  
}
```

W konstruktorze (niestatycznym) klasy `Student` zwiększymy wartość o jeden pola `ostatniNumerIndeksu` i przypiszemy ją polu `numerIndeksu`. Na koniec zapiszemy nową wartość pola `ostatniNumerIndeksu` do pliku. Konstruktor po dodaniu kodu powinien wyglądać następująco:

```
public Student(string imie, string nazwisko) {  
    Imie = imie;  
    Nazwisko = nazwisko;  
    ostatniNumerIndeksu++;  
    numerIndeksu = ostatniNumerIndeksu;  
    BinaryWriter bw = null;  
    try {  
        bw = new BinaryWriter(File.Open(NAZWAPLIKU, FileMode.Truncate));  
        bw.Write(ostatniNumerIndeksu);  
    }  
    finally {  
        if (bw != null)  
            bw.Close();  
    }  
}
```

Dodanie możliwości współpracy z klasą `Console` do klasy `Student`

Klasa `Student` będzie wykorzystywana w aplikacji konsolowej. Musimy zatem umożliwić inicjalizację, nadanie wartości poszczególnym polom i wyświetlenie wartości poszczególnych pól

klasy `Student` przy pomocy klasy `Console`. Tym razem zaimplementujemy to przy pomocy metod rozszerzających. Metody rozszerzające, jak już było wspomniane, mogą być definiowane tylko w klasach statycznych. Utwórzmy zatem taką klasę:

```
public static class StudentDlaKonsoli {  
}
```

Do wcześniej utworzonej klasy dodajmy metodę, która pobierze od użytkownika konieczne informacje, utworzy obiekt klasy `Student` i przekaże go do programu:

```
public static Student UtworzStudenta() {  
    Console.Write("Podaj nazwisko: ");  
    string nazwisko = Console.ReadLine();  
    Console.Write("Podaj imię: ");  
    string imie = Console.ReadLine();  
    return new Student(imie, nazwisko);  
}
```

Dodajmy następnie metody, przy pomocy których będziemy mogli zmodyfikować obiekt klasy `Student`:

```
public static void ZmienNazwisko(this Student s) {  
    Console.Write("Podaj nazwisko: ");  
    s.Nazwisko = Console.ReadLine();  
}  
  
public static void ZmienImie(this Student s) {  
    Console.Write("Podaj imię: ");  
    s.Imie = Console.ReadLine();  
}  
  
public static void ZmienDaneStudenta(this Student s) {  
    s.ZmienNazwisko();  
    s.ZmienImie();  
}
```

Na zakończenie dodajmy metodę, przy pomocy której będziemy mogli wypisać informacje na temat danego studenta:

```
public static void WypiszWszystkieInformacje(this Student s) {  
    Console.WriteLine("Pan(i) {0} {1} numer indeksu {2}",  
        s.Imie, s.Nazwisko, s.NumerIndeksu);  
}
```

Utworzenie programu testującego klasę `Student`

Kod testujący klasę `Student` może wyglądać następująco:

```
static void Main(string[] args) {  
    StudentDlaKonsoli s1 = new StudentDlaKonsoli();  
    s1.WypiszStudenta();  
    s1.ZmienImie();  
    s1.ZmienNazwisko();  
  
    Student s2 = s1.Student;  
    Console.WriteLine("Pan(i) {0} {1} numer indeksu {2}",  
        s2.Imie, s2.Nazwisko, s2.NumerIndeksu);  
  
    Student s3 = new Student("Jan", "Kowalski", 1234);  
    StudentDlaKonsoli s4 = new StudentDlaKonsoli(s3);  
    s4.WypiszStudenta();  
}
```

```
    Console.ReadKey();  
}
```

Gotowe rozwiązanie powyższych przykładów znajduje się w katalogu Demo\Modul04.

Porady praktyczne

- W przypadku gdy w metodzie nie odwołujesz się bezpośrednio do żadnej zwykłej składowej (składowej niestatycznej), rozważ czy nie zdefiniować jej jako metody statycznej. Przy podejmowaniu decyzji, czy metodę zaimplementować jako statyczną lub nie, powinieneś uwzględnić, czy metoda nie powinna być wirtualna co wyklucza jej statyczność. Powinieneś również przemyśleć ewentualne zmiany implementacji metody - czy w przyszłości nie jest prawdopodobne, że metoda będzie jednak potrzebować dostępu do zwykłych pól.
- Do metod statycznych nie jest przekazywany parametr `this`, dlatego nie można w ich ciele odwoływać się bezpośrednio do niestycznych składowych danej klasy.
- Pola `const` są również niejawnie polami statycznymi.
- Pole statyczne tylko do odczytu nie jest równoważne polu `const`. Wartość pola `const` musi być znana w czasie kompilacji. Polu `readonly` możesz nadać wartość w czasie działania aplikacji, np. wczytując wartość z pliku konfiguracyjnego.
- Jako składowe statyczne możesz definiować pola, metody, właściwości oraz zdarzenia.
- Staraj się definiować składowe statyczne w jednym miejscu w danej klasie lub strukturze. Ułatwia to analizę Twojego kodu.
- Zmienne lokalne metody nie mogą być definiowane jako `static`. Spowoduje to błąd kompilacji.
- Pamiętaj, konstruktor statyczny jest to metoda, która jest wywoływana przed utworzeniem obiektu i przed odwołaniem się do którejkolwiek ze składowych statycznych danej klasy.
- Konstruktor statycznego nie można wywołać jawnie.
- Konstruktor statycznego nie można przeciążyć, jest on metodą bezparametrową.
- Definiując konstruktor statyczny nie wolno określać modyfikatora dostępu.
- Konstruktor statyczny jest metodą statyczną, dlatego również nie można w jego ciele odwoływać się do niestycznych składowych.
- Pola statyczne inicjalizuj w miejscu ich definicji, a nie w konstruktorze statycznym, chyba że ważna jest kolejność inicjalizacji lub trzeba wykonać dodatkowe instrukcje. Kod Twojej klasy będzie bardziej zrozumiały.
- Typy strukturalne również mogą mieć składowe statyczne, zarówno pola, jak i metody statyczne. Dla struktur możesz również definiować konstruktor statyczny.
- Pamiętaj, nie można zdefiniować zmiennej, której typ określony jest przez klasę statyczną. Nie można utworzyć również obiektu klasy statycznej.
- Pola i metody klas statycznych zastępują niejako zmienne globalne i funkcje globalne, które możemy definiować w innych językach programowania, np. w C++.
- Struktur nie wolno definiować jako statycznych.
- Nie umieszczaj wszystkich swoich metod globalnych w pojedynczej klasie statycznej. Klasa statyczna, jak każda klasa, powinna zawierać składowe powiązane ze sobą, realizujące określone zadanie.
- Metody rozszerzające można definiować tylko w klasach statycznych.
- Pamiętaj, przy pomocy metod rozszerzających możemy dodać metody do istniejących typów bez konieczności ponownej kompilacji ich kodów źródłowych.
- Przy pomocy metod rozszerzających do klasy nie możemy dodać: pola, właściwości i zdarzenia.
- Metody rozszerzające mają dostęp tylko do publicznych składowych klasy, którą rozszerzają.
- Przy pomocy wzorca Singleton możemy imitować w języku C# zmienne globalne.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- wiesz, do czego służą pola statyczne i potrafisz definiować je we własnych klasach
- wiesz, do czego służą metody statyczne
- potrafisz definiować własne metody statyczne
- rozumiesz, do czego służy konstruktor statyczny
- wiesz jak definiować konstruktor statyczny
- rozumiesz, do czego służą klasy statyczne
- potrafisz definiować własne klasy statyczne
- wiesz, co to są metody rozszerzające
- potrafisz definiować własne metody rozszerzające
- znasz ograniczenia metod rozszerzających
- wiesz, do czego służy wzorzec projektowy o nazwie singleton

Dodatkowe źródła informacji

1. Daniel Solis, *Illustrated C# 2008*, Apress, 2008

Książka dla tych wszystkich którzy pragną nauczyć się tworzyć programy w języku C#. Zawiera dokładne omówienie tego języka.

2. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka dla programistów chcących nauczyć się programować w języku C#.

3. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad, jak powinniśmy pisać swoje programy.

4. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Wzorce projektowe Wydanie II*, WNT, 2008

Książka, która wprowadziła pojęcie wzorców projektowych do informatyki. Lektura obowiązkowa dla każdego, kto chce poznać temat wzorców projektowych.

5. Steven John Metsker, *C#. Wzorce projektowe*, Helion, 2005

Książka jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia, jak wykorzystać cechy języka C# do tworzenia poprawnego kodu poprzez zastosowanie wzorców.

6. Judith Bishop, *C# 3.0 Design Patterns*, O'Reilly Media, Inc. 2008

Książka, podobnie jak poprzednia, jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia również nowe cechy języka C#.

7. Codeguru, <http://www.codeguru.pl>

Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.



Laboratorium podstawowe

Problem 1 (czas realizacji 45 minut)

Twoja firma opracowuje program dla pewnego banku. Twoim zadaniem jest dokończenie i przetestowanie klasy Konto. Do klasy Konto musisz dodać pole oznaczające jego numer. Numer ten musi być unikalny dla każdego konta. Musisz również dodać pola oznaczające oprocentowanie konta oraz dopuszczalny debet. Dla wszystkich kont pola te mają przyjmować jednakowe wartości. Zmiana wartości tych pól powinna być możliwa jedynie po podaniu odpowiedniego hasła. Klasa Konto będzie wykorzystywana w aplikacji konsolowej, powinieneś zatem zaimplementować metody rozszerzające umożliwiające wyświetlanie informacji o obiekcie klasy Konto i pobieranie nowych wartości dla jego pól przy pomocy klasy Console.

Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition	<ul style="list-style-type: none"> Otwórz Visual C# 2008 Express Edition. Z menu wybierz File -> New Project. Z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz Bank. Kliknij OK. Z menu wybierz File -> Save Kadry. W polu Location wybierz folder w którym będzie zapisany projekt. Zaznacz pole wyboru Create directory for solution. W polu Solution Name wpisz Modul04. Naciśnij przycisk Save.
2. Do projektu Bank dodaj plik z implementacją klasy Konto	<ul style="list-style-type: none"> Z menu wybierz Project -> Add Existing Item. W oknie dialogowym Add ExistingItem – Bank wybierz plik Konto.cs, który znajduje się w katalogu Kurs\Lab\Start\Modul04, gdzie Kurs jest katalogiem, w którym zainstalowano pliki kursu. Kliknij Add.
3. Do klasy Konto dodaj pole oznaczające numer konta i zaimplementuj jego automatyczne nadawanie	<ul style="list-style-type: none"> Przejdź do pliku Konta.cs. Na górze pliku zaimportuj przestrzeń nazw System.IO: <pre>using System.IO;</pre> Do klasy Konto dodaj pole const typu string, które będzie przechowywać ścieżkę dostępu i nazwę pliku zawierającego numer ostatnio dodanego konta: <pre>private const string NAZWAPLIKU = "numer.dat";</pre> Do klasy Konto dodaj prywatne pole reprezentujące numer konta i publiczną właściwość tylko do odczytu zwracającą jego wartość: <pre>readonly private int numerKonta; public int NumerKonta { get { return numerKonta; } }</pre> Do klasy Konto dodaj statyczne prywatne pole przechowujące numer ostatnio dodanego konta: <pre>private static int ostatniNumerKonta;</pre> Do klasy Konto dodaj statyczny konstruktor i w odpowiedni sposób zainicjalizuj w nim pole ostatniNumerKonta:

	<pre>static Konto() { if(File.Exists(NAZWAPLIKU)) { BinaryReader br = null; try { br = new BinaryReader(File.Open(NAZWAPLIKU, FileMode.Open)); ostatniNumerKonta = br.ReadInt32(); } finally { if(br != null) br.Close(); } } else { ostatniNumerKonta = 0; } }</pre> <ul style="list-style-type: none"> • Zmodyfikuj w odpowiedni sposób konstruktor klasy Konto przyjmujący trzy parametry w celu inicjalizacji pola numerKonta. Po modyfikacji konstruktor powinien wyglądać następująco: <pre>public Konto(Osoba wlasciciel, decimal saldoPocztkowe, int pin) { Wlasciciel = wlasciciel; this.saldo = saldoPocztkowe; this.pin = pin; ostatniNumerKonta++; numerKonta = ostatniNumerKonta; BinaryWriter bw = null; try { bw = new BinaryWriter(File.Open(NAZWAPLIKU, FileMode.Create)); bw.Write(ostatniNumerKonta); } finally { if (bw != null) bw.Close(); } }</pre>
<p>4. Do klasy Konto dodaj pola reprezentujące debet i oprocentowanie konta oraz metody do zmiany ich wartości</p>	<ul style="list-style-type: none"> • Do klasy Konto dodaj prywatne statyczne pole reprezentujące oprocentowanie konta i publiczną statyczną właściwość tylko do odczytu zwracającą jego wartość. Pole zainicjalizuj wartością 0,05: <pre>private static decimal oprocentowanie = 0.05m; public static decimal Oprocentowanie { get { return oprocentowanie; } }</pre> • Do klasy Konto dodaj prywatne statyczne pole reprezentujące dopuszczalny debet na koncie i publiczną statyczną właściwość tylko do odczytu zwracającą jego wartość. Pole zainicjalizuj wartością 1000: <pre>private static decimal debet = 1000; public static decimal Debet { get { return debet; } }</pre> • Do klasy Konto dodaj pole statyczne typu string, które będzie przechowywać hasło wymagane przy zmianie wartości pól oprocentowanie i debet. Zainicjalizuj je wartością P@ssw0rd: <pre>private static string haslo = "P@ssw0rd";</pre> • Do klasy Konto dodaj publiczną statyczną metodę odpowiedzialną za

	<p>zmianę wartości pola haslo:</p> <pre>public static bool ZmienHaslo(string stareHaslo, string noweHaslo){ if(stareHaslo == haslo) { haslo = noweHaslo; return true; } return false; }</pre> <p> Czy w metodzie ZmienHaslo można odwołać się bezpośrednio do pola pin? Wyjaśnij dlaczego?</p> <ul style="list-style-type: none"> Do klasy Konto dodaj publiczną statyczną metodę odpowiedzialną za zmianę wartości pola debet: <pre>public static bool ZmienDebet(string haslo, decimal nowyDebet) { if(Konto.haslo == haslo) { debet= nowyDebet; return true; } return false; }</pre> <p> Wyjaśnij dlaczego w metodzie ZmienDebet używamy konstrukcji Konto.haslo, a nie konstrukcji this.haslo?</p> <ul style="list-style-type: none"> Do klasy Konto dodaj publiczną statyczną metodę odpowiedzialną za zmianę wartości pola oprocentowanie: <pre>public static bool ZmienOprocentowanie(string haslo, decimal noweOprocentowanie) { if(Konto.haslo == haslo) { oprocentowanie = noweOprocentowanie; return true; } return false; }</pre>
5. Zmień kod metody DokonajWyplaty , aby uwzględniała dopuszczalny debet	<ul style="list-style-type: none"> Przejdź do metody DokonajWyplaty. Zmień warunek sprawdzający, czy możliwe jest popranie z konta pieniędzy. Po modyfikacji kod metody DokonajWyplaty powinien wyglądać następująco: <pre>public bool DokonajWyplaty(decimal kwota, int pin) { if (!SprawdzPin(pin) saldo + debet < kwota) return false; saldo -= kwota; return true; }</pre>
6. Do klasy Konto dodaj możliwość współpracy z konsolą przy pomocy metod rozszerzających	<ul style="list-style-type: none"> Z menu wybierz Project -> Add Class. W oknie dialogowym Add New Item – Bank z listy Visual Studio installed templates wybierz Class. W polu Name wpisz KontoDlaKonsoli. Kliknij OK. Uczyń nowo utworzoną klasę KontoDlaKonsoli klasą statyczną przez dodanie słowa static przed słowem class: <pre>static class KontoDlaKonsoli { }</pre> <ul style="list-style-type: none"> Do klasy KontoDlaKonsoli dodaj dwie pomocnicze prywatne metody. Służyć będą do pobierania wartości typu int i decimal: <pre>private static int pobierzLiczbeInt(string s) {</pre>

```

int pin;
do {
    Console.Write(s);
}
while (!int.TryParse(Console.ReadLine(), out pin));
return pin;
}

private static decimal pobierzLiczbeDecimal(string s) {
    decimal kwota;
    do {
        Console.Write(s);
    }
    while (!decimal.TryParse(Console.ReadLine(), out kwota));
    return kwota;
}

```

- Do klasy **KontoDlaKonsoli** dodaj publiczną metodę, która pobierze od użytkownika konieczną informację do utworzenia obiektu klasy **Konto**., a następnie utworzony obiekt zwróci do programu:

```

public static Konto UtworzKonto() {
    Console.Write("Podaj imię: ");
    string imie = Console.ReadLine();
    Console.Write("Podaj nazwisko: ");
    string nazwisko = Console.ReadLine();
    decimal saldo = pobierzLiczbeDecimal("Podaj saldo początkowe: ");
    int pin = pobierzLiczbeInt("Podaj pin: ");
    return new Konto(new Osoba(imie,nazwisko),saldo,pin);
}

```

- Do klasy **KontoDlaKonsoli** dodaj publiczną metodę, przy pomocy której będzie można dokonać wpłaty na konto:

```

public static void Wplac(this Konto k) {
    decimal kwota =
        pobierzLiczbeDecimal("Podaj kwotę, którą chcesz wpłacić: ");
    k.DokonajWplaty(kwota);
}

```

- Do klasy **KontoDlaKonsoli** dodaj publiczną metodę, przy pomocy której będzie można dokonać wypłaty z konta:

```

public static void Wyplac(this Konto k) {
    decimal kwota =
        pobierzLiczbeDecimal("Podaj kwotę, którą chcesz wypłacić: ");
    int pin = pobierzLiczbeInt("Podaj pin: ");
    if (k.DokonajWypłaty(kwota,pin)) {
        Console.WriteLine("Proszę przejść do kasy po pieniądze");
    }
    else {
        Console.WriteLine(
            "Niestety, ta operacja nie może być wykonana");
    }
}

```

- Do klasy **KontoDlaKonsoli** dodaj publiczną metodę, przy pomocy której będzie można obejrzeć informacje o koncie:

```

public static void WypiszInformacjeOKoncie(this Konto k){
    int pin = pobierzLiczbeInt("Podaj pin: ");
    Console.WriteLine(k.PobierzInformacje(pin));
    Console.WriteLine("Informacja dla konta o numerze: {0}",
        k.NumerKonta);
}

```

- Do klasy **KontoDlaKonsoli** dodaj publiczną metodę, przy pomocy której

będzie można dokonać zmiany pinu:

```
public static void ZmienPin2(this Konto k) {
    int stary = pobierzLiczbeInt("Podaj stary pin: ");
    int nowy = pobierzLiczbeInt("Podaj nowy pin: ");
    int nowy2 = pobierzLiczbeInt("Powtórz nowy pin: ");
    if (nowy == nowy2) {
        if(k.ZmienPin(nowy, stary) ) {
            Console.WriteLine("Hasło zostało zmienione");
            return;
        }
    }
    Console.WriteLine("Wprowadzone piny są niezgodne");
}
```



Jakie były by konsekwencje, gdyby powyższą metodę nazwać **ZmienPin**, a nie **ZmienPin2**?

- Do klasy **KontoDlaKonsoli** dodaj publiczną metodę, przy pomocy której będzie można dokonać zmiany hasła:

```
public static void ZmienHaslo() {
    Console.Write("Podaj stare hasło: ");
    string stare = Console.ReadLine();
    Console.Write("Podaj nowe hasło: ");
    string nowe = Console.ReadLine();
    Console.Write("Powtórz nowe hasło: ");
    string nowe2 = Console.ReadLine();
    if (nowe == nowe2) {
        if (Konto.ZmienHaslo(stare, nowe)) {
            Console.WriteLine("Hasło zostało zmienione");
            return;
        }
    }
    Console.WriteLine("Wprowadzone hasła są niezgodne");
}
```

- Do klasy **KontoDlaKonsoli** dodaj publiczną metodę, przy pomocy której będzie można wyświetlić informację o dopuszczalnym debecie:

```
public static void WypiszDebet() {
    Console.WriteLine(
        "Aktualny maksymalny debet na koncie wynosi {0}",
        Konto.Debet);
}
```

- Do klasy **KontoDlaKonsoli** dodaj publiczną metodę, przy pomocy której będzie można wyświetlić informację o oprocentowaniu:

```
public static void WypiszOprocentowanie() {
    Console.WriteLine("Aktualne oprocentowanie kont wynosi {0}",
        Konto.Oprocentowanie);
}
```

- Do klasy **KontoDlaKonsoli** dodaj publiczną metodę, przy pomocy której będziemy mogli dokonać zmiany oprocentowania:

```
public static void ZmienOprocentowanie() {
    Console.Write("Podaj hasło: ");
    string haslo = Console.ReadLine();
    decimal oprocentowanie =
        pobierzLiczbeDecimal("Podaj nowe oprocentowanie: ");
    if (Konto.ZmienOprocentowanie(haslo, oprocentowanie)) {
        Console.WriteLine("Oprocentowanie zostało zmienione");
    }
    else {
        Console.WriteLine("Operacja została anulowana");
    }
}
```

	<pre> } } </pre> <ul style="list-style-type: none"> Do klasy KontoDlaKonsoli dodaj publiczną metodę, przy pomocy której będziemy mogli dokonać zmiany maksymalnego dopuszczalnego debetu: <pre> public static void ZmienMaksymalnyDebet() { Console.WriteLine("Podaj hasło: "); string haslo = Console.ReadLine(); decimal maxDebet = pobierzLiczbeDecimal("Podaj nowy maksymalny dopuszczalny debet: "); if (Konto.ZmienDebet(haslo, maxDebet)) { Console.WriteLine("Maksymalny dopuszczalny debet został zmieniony"); } else { Console.WriteLine("Operacja została anulowana"); } } } </pre>
7. Przetestuj klasę Konto i klasę KontoDlaKonsoli	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Do metody Main dodaj kod, który przetestuje, czy klasy Konto i KontoDlaKonsoli działają prawidłowo: <pre> Konto k1 = KontoDlaKonsoli.UtworzKonto(); k1.Wplac(); k1.Wyplac(); k1.WypiszInformacjeOKoncie(); k1.ZmienPin2(); KontoDlaKonsoli.WypiszDebet(); KontoDlaKonsoli.WypiszOprocentowanie(); KontoDlaKonsoli.ZmienHaslo(); KontoDlaKonsoli.ZmienMaksymalnyDebet(); KontoDlaKonsoli.ZmienOprocentowanie(); Console.ReadKey(); </pre>
8. Skompiluj i uruchom program	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 90 min)

Twoim zadaniem jest utworzenie programu, który będzie wczytywał plik tekstowy i analizował go pod względem liczby wystąpień określonych znaków, fraz itp. Firma w której pracujesz, zajmuje się obróbką tekstów, więc dodatkowym wymaganiem jest, abyś utworzył bibliotekę metod analizujących łańcuch znaków. Biblioteka ta powinna zawierać następujące metody:

- Metodę obliczającą liczbę wystąpień określonego znaku.
- Metodę obliczającą liczbę wystąpień określonego ciągu znaków.
- Metodę obliczającą liczbę wystąpień określonej litery. Metoda powinna mieć dwie wersje, jedną uwzględniającą wielkość litery, drugą obliczającą ilość wystąpień danej litery bez względu na jej wielkość.
- Metodę obliczającą liczbę wystąpień określonej frazy. Metoda powinna mieć dwie wersje, jedną uwzględniającą wielkość liter, drugą obliczającą ilość wystąpień danej frazy bez względu na wielkość liter ją tworzących.
- Metodę obliczającą liczbę wystąpień liter.
- Metodę obliczającą liczbę wystąpień samogłosek.
- Metodę obliczającą liczbę wystąpień spółgłosek.
- Metodę obliczającą liczbę wystąpień cyfr.
- Metodę obliczającą liczbę wystąpień znaków alfanumerycznych (cyfr i liter).
- Metodę obliczającą liczbę wystąpień znaków niealfanumerycznych.
- Metodę obliczającą liczbę wystąpień białych znaków (spacja, tabulacja i przejście do nowej linii).
- Metodę obliczającą liczbę wystąpień małych liter.
- Metodę obliczającą liczbę wystąpień wielkich liter.

Zadecydowano, że metody te muszą być zaimplementowane jako metody rozszerzające typ `string`, aby inni programiści mogli z nich korzystać w łatwy, intuicyjny i przyjemny sposób.

Wskazówka: do sprawdzania, czy dany znak jest cyfrą, literą itp., możesz użyć metod statycznych typu `char.IsDigit`, `char.IsLetter` itp.

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 5

Wersja 2

Przeciążenie operatorów

Spis treści

Przeciążenie operatorów	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	8
Porady praktyczne	10
Uwagi dla studenta	11
Dodatkowe źródła informacji	11
Laboratorium podstawowe	12
Problem 1 (czas realizacji 45 minut)	12
Laboratorium rozszerzone	19
Zadanie 1 (czas realizacji 90 min)	19

Informacje o module

Opis modułu

W tym module zapoznasz się z pojęciem przeciążania operatorów. Dowiesz się, które operatory można przeciążyć w języku C#. Nauczysz się, jak definiować metody, które będą implementowały żądany operator. Poznasz szczegóły dotyczące implementacji poszczególnych operatorów. Dowiesz się, jak można zdefiniować w języku C# metody, które będą zamieniać obiekt jednego typu na obiekt innego typu.

Cel modułu

Celem modułu jest pokazanie mechanizmu zwanego przeciążeniem operatorów i jego możliwości użycia w języku C#.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- znał pojęcie przeciążenie operatorów
- wiedział, które operatory można przeciążyć
- wiedział, jak definiować własne operatory dla swoich typów
- wiedział, jak definiować operatory konwersji dla własnych typów

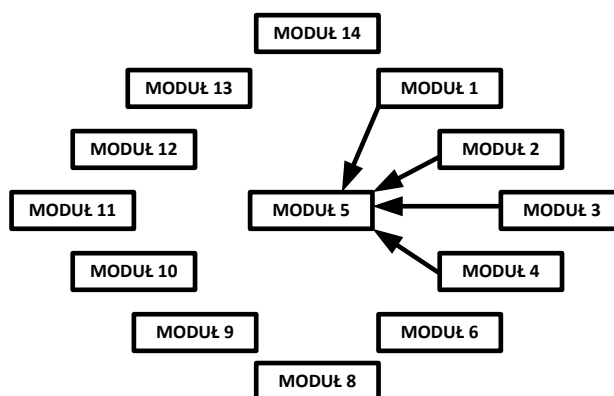
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinieneś:

- potrafić definiować klasę
- wiedzieć, co to jest pole i co to jest metoda
- wiedzieć, co to są składniki statyczne i potrafić je definiować
- znać operatory, którymi można posługiwać się w języku C#

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na Rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w modułach „Pojęcie klasy”, „Konstruktor”, „Właściwości i indeksatory”, oraz „Składowe statyczne”.



Rys. 7 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Korzystając z typów wbudowanych, zapewne niejednokrotnie korzystałeś z operatorów do wykonania podstawowych operacji. Podstawową zaletą operatorów jest ich intuicyjność. Większość wyrażeń utworzonych przy pomocy podstawowych operatorów jest dla wszystkich zrozumiała. Możesz zadać sobie pytanie, czy operatory można stosować z typami przez siebie zdefiniowanymi. W swoich programach na pewno będziesz niejednokrotnie potrzebował lub nawet już używał wektorów, macierzy czy liczb zespolonych. Możesz przecież w łatwy sposób stworzyć strukturę reprezentującą np. wektor czy liczbę zespoloną. Dla dwóch wektorów lub liczb zespolonych symbole $+$, $-$, czy $*$ w matematyce są jasno zdefiniowane. Dużo łatwiej w programie jest napisać:

```
wektor1 + wektor2
```

niż:

```
Wektor.Dodaj(wektor1,wektor2)
```

Pierwsze wyrażenie wydaje się również bardziej naturalne. Podobnie dla naszych klas możemy chcieć nadać znaczenie operatorom relacyjnym. Można porównać np. dwa obiekty klasy *Samochod* pod względem ceny, a użyć do tego symbolu znaku mniejszości $<$. Jeśli chcesz wiedzieć jak, można zdefiniować operatory, których będziesz mógł używać z obiektami swoich klas, zapoznaj się z treścią tego modułu.

Podstawy teoretyczne

Przeciążanie operatorów (ang. *operator overloading*) lub inaczej *przeładowywanie operatorów* jest pewnym rodzajem przeciążania (przeładowania) metod. Polega ono na tym, że operator może mieć różną funkcjonalność w zależności od typu użytych argumentów (operandów). Dzięki temu, że język C# wspiera mechanizm przeciążenia operatorów, możemy definiować operatory współpracujące ze zdefiniowanymi przez nas typami. Niestety nie wszystkie operatory można przeciążyć. W języku C# można przeciążyć następujące operatory:

- operatory jednoargumentowe: $+$, $-$, $!$, \sim , $++$, $--$, `true`, `false`
- operatory relacji: $==$, $!=$, $<$, $>$, $<=$, $>=$
- pozostałe operatory dwuargumentowe: $+$, $-$, $*$, $/$, $\%$, $\&$, $|$, $^$, $<<$, $>>$

Złożonych operatorów przypisania: $+=$, $-=$, $*=$, $/=$, $\%=$, $\&=$, $|=$, $^=$, $<<=$ i $>>=$ nie można przeciążyć bezpośrednio, ale można otrzymać ich wersje dla własnych typów niejako w prezencie, przeciążając odpowiednie operatory binarne $+$, $-$, $*$, $/$, $\%$, $\&$, $|$, $^$, $<<$ i $>>$.

Zakładając, że chcemy przeciążyć operator $+=$, wystarczy przeciążyć dwuargumentowy operator dodawania $+$ i wtedy wyrażenie:

```
a += b
```

zostanie przekształcone na:

```
a = a + b
```

Przypomnijmy, że operator przypisania jest automatycznie generowany dla wszystkich typów (oba argumenty muszą być tego samego typu) i w przypadku typów referencyjnych oznacza skopiowanie odwołań, a w przypadkach typów bezpośrednich wykonywana jest kopia bit po bicie.

Podobnie jak złożonych operatorów przypisania, nie można bezpośrednio przeciążyć operatorów $\&\&$ oraz $|$. W celu otrzymania przeciążonej wersji tych operatorów dla własnych typów, musimy przeciążyć operatory `true`, `false`, $|$ i $\&$. Wartość wyrażeń z użyciem tych operatorów jest obliczana według wzorów:

- `x || y ::= true(x) ? x : x | y`
- `x && y ::= false(x) ? x : x & y`

Należy zwrócić uwagę, że wyrażenia `true(x)` i `false(x)` nie są poprawnym sposobem wywołania odpowiednio operatorów `true` i `false` i pełnią tylko rolę pomocniczą. Więcej na temat operatorów `true` i `false` w dalszej części tego modułu.

Innym operatorem, którego nie można przeciążyć w języku C#, jest operator `[]`. Możemy jednak rozważyć, czy indeksator, omówiony w module 3 tego kursu, nie jest innym sposobem przeciążenia operatora `[]`.

Pod koniec tego rozdziału zostaną omówione operatory konwersji. Operatory te są przeciążeniem operatora konwersji `()`.

Poznaliśmy, które operatory można przeciążyć. Teraz pokażemy, jak w języku C# definiuje się własne operatory. W celu przeciążenia operatora musimy zdefiniować odpowiednią metodę. Metoda ta musi spełniać następujące warunki:

- musi być metodą publiczną
- musi być metodą statyczną
- typ co najmniej jednego z argumentów lub typ wartości zwracanej musi być określony przez klasę lub strukturę, w której dana metoda jest zdefiniowana
- metoda musi coś zwracać, typ wartości przekazywanej nie może być `void`
- wszystkie argumenty metody muszą być przekazywane przez wartość
- nazwa metody musi mieć format: `operator op`, gdzie `op` jest symbolem operatora

W dalszej części modułu przeciążenie poszczególnych operatorów będziemy omawiać korzystając z następującej struktury:

```
struct MojaLiczba {  
    public int Liczba;  
}
```

Mechanizm przeciążenia operatorów jest taki sam dla klas jak i dla struktur, więc nie ma znaczenia, czy w przykładach użyjemy klasy czy struktury.

Operatory jednoargumentowe

W przypadku operatorów unarnych, typ jedyne argumentu przesyłanego do metody definiującej dany operator musi być określony przez klasę lub strukturę, w której dana metoda jest zdefiniowana.

W przypadku operatorów: `+`, `-`, `!` i `~`, typem zwracanym może być dowolny typ, oczywiście poza `void`. Przykładową implementację operatorów `+`, `-`, `!` i `~` dla struktury `MojaLiczba` umieszczono poniżej:

```
//operator ! oblicza silnię  
public static long operator!(MojaLiczba x) {  
    if(x.Liczba < 0)  
        throw new ArgumentException(  
            "Silnia nie jest zdefiniowana dla liczb ujemnych");  
    long silnia = 1;  
    for (int i = 2; i <= x.Liczba; i++)  
        silnia *= i;  
    return silnia;  
}  
  
//operator ~ oblicz wartość odwrotną  
public static double operator ~(MojaLiczba x) {  
    return 1.0 / x.Liczba;  
}
```

```
}

public static int operator +(MojaLiczba x) {
    return x.Liczba;
}

public static int operator -(MojaLiczba x) {
    return -x.Liczba;
}
```

Powyżej zdefiniowane operatory możemy używać w następujący sposób:

```
MojaLiczba x1 = new MojaLiczba();
x1.Liczba = 5;
Console.WriteLine(!x1);    //zostanie wypisane 120
Console.WriteLine(~x1);    //zostanie wypisane 0,2
Console.WriteLine(+x1);    //zostanie wypisane 5
Console.WriteLine(-x1);    //zostanie wypisane -5
```

W przypadku metod definiujących operatory ++ i --, typ wartości przekazywanej musi być określony przez klasę lub strukturę, w której dana metoda jest zdefiniowana. Jeżeli operator jest zdefiniowany dla klasy, to typem wartości zwracanej może być również klasa pochodna po danej klasie. Na temat klas pochodnych można przeczytać w module 6 tego kursu. Operatory ++ i -- dla struktury `MojaLiczba` możemy zdefiniować w następujący sposób:

```
public static MojaLiczba operator ++(MojaLiczba x) {
    x.Liczba++;
    return x;
}

public static MojaLiczba operator --(MojaLiczba x) {
    x.Liczba--;
    return x;
}
```

Powyższe metody definiują odpowiednio operator inkrementacji i dekrementacji zarówno w wersji przyrostkowej, jak i przedrostkowej.

Operatory `true` i `false` musimy przeciążyć jednocześnie, to znaczy, że jeśli chcemy przeciążyć operator `true`, to musimy także przeciążyć operator `false` i na odwrót. Metody definiujące te operatory jako typ wartości zwracanej muszą mieć typ logiczny. Zmienne typu, dla którego przeciążyliśmy operatory, mogą występować jako warunek w instrukcji warunkowej lub w pętlach. Przykładowa implementacja tych operatorów jest umieszczona poniżej:

```
public static bool operator true(MojaLiczba x) {
    return x.Liczba != 0;
}

public static bool operator false(MojaLiczba x) {
    return x.Liczba == 0;
}
```

Mając zdefiniowane w powyższy sposób operatory `true` i `false`, w celu sprawdzenia, czy zmienna typu `MojaLiczba` o nazwie `x1` zawiera wartość 0, możemy użyć następującego kodu:

```
if (x1) {
    Console.WriteLine("Różna od zera");
}
else {
    Console.WriteLine("Równa zero");
}
```

Operatory relacji

Operatory relacji muszą być przeciążane parami. To znaczy, że jeśli przeciążymy operator większości `>`, to musimy także przeciążyć operator mniejszości `<`. Podobnie musimy przeciążyć razem operator `==` i `!=` oraz `<=` i `>=`. Przykładową implementację operatorów relacji dla struktury `MojaLiczba` umieszczono poniżej:

```
public static bool operator <(MojaLiczba x, int y) {
    return x.Liczba < y;
}

public static bool operator >(MojaLiczba x, int y) {
    return x.Liczba > y;
}

public static bool operator <(MojaLiczba x, MojaLiczba y) {
    return x.Liczba < y.Liczba;
}

public static bool operator >(MojaLiczba x, MojaLiczba y) {
    return x.Liczba > y.Liczba;
}
```

Warto zwrócić uwagę, że metody implementujące operatory relacji niekoniecznie muszą zwracać wartość typu logicznego, choć stosowanie innego typ nie jest zalecane.

Zajmijmy się teraz dokładniej operatorami `==` i `!=`. Dla typów referencyjnych operatory te mają predefiniowane znaczenie. Operatory te, jeżeli nie przeciążymy ich, sprawdzają, czy zmienne odwołują się do tego samego obiektu, tj. sprawdzają odpowiednio równość lub „różność” referencji.

Zaleca się (kompilator generuje nawet ostrzeżenie), że jeżeli zdecydujemy się przeciążyć operatory `==` i `!=`, powinniśmy również nadpisać metody typu `Object: Equals` i `GetHashCode`. Szczegóły na temat nadpisywania metod typu `Object` można znaleźć w module 8.

W ogóle zaleca się, aby do porównywania obiektów lub zmiennych strukturalnych implementować metody interfejsu `ICompareable` oraz `IComparer`. Interfejsy opisane są w modułach 9 i 11.

Powyższe zalecenia spowodowane są tym, że nie we wszystkich językach programowania, w których możemy pisać programy na platformę .NET, możemy stosować operatory.

Operatory binarne

W przypadku operatorów binarnych (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<` i `>>`) warunkiem koniecznym jest, aby typ jednego z argumentów metody definiującej operator był określony przez klasę lub strukturę, w której dana metoda jest zdefiniowana. Dodatkowe wymagania mają operatory `<<` i `>>`. W ich przypadku pierwszy argument musi być typu określonego przez klasę lub strukturę, dla której definiujemy dany operator. Drugi argument musi być typu `int`.

Przykładową implementację operatorów dwuargumentowych dla struktury `MojaLiczba` umieszczono poniżej:

```
public static MojaLiczba operator +(MojaLiczba x, MojaLiczba y) {
    MojaLiczba z;
    z.Liczba = x.Liczba + y.Liczba;
    return z;
}

public static long operator ^(MojaLiczba x, int n) {
    long potega = 1;
    for (int i = 1; i <= (n > 0 ? n : -n); i++)
```

```
        potega *= x.Liczba;
    if (n < 0)
        return 1 / potega;
    return potega;
}
```

Operatory konwersji

Przypomnijmy, że konwersja jest to przekształcenie obiektu jednego typu, na obiekt drugiego typu. W celu zdefiniowania własnej konwersji, definiujemy następującą metodę:

```
public static <rodzaj_konwersji> operator typ_docelowy (
    typ_konwertowany nazwa_argumentu){
    ciało_metody
}
```

<rodzaj_konwersji> ::= implicite | explicit

Typ docelowy jest jednocześnie typem wartości zwracanej przez metodę realizującą funkcję operatora.

Oczywiście jeden z typów: typ docelowy lub typ konwertowany musi być określony przez klasę lub strukturę, w której dana metoda jest zdefiniowana.

Rodzaj konwersji określa nam, czy dana konwersja może być wywołana niejawnie (automatycznie) przez kompilator lub czy programista musi jawnie użyć operatora konwersji. Rozważmy to na przykładzie naszej struktury `MojaLiczba`. Załóżmy, że mamy zdefiniowane następujące metody:

```
public static implicit operator MojaLiczba(int x) {
    MojaLiczba y;
    y.Liczba = x;
    return y;
}

public static explicit operator MojaLiczba(string x) {
    MojaLiczba y;
    y.Liczba = int.Parse(x);
    return y;
}
```

Pierwsza metoda definiuje konwersję niejawną z typu `int` na typ `MojaLiczba`. W programie możemy więc stosować zmienne lub literały typu `int` wszędzie tam, gdzie spodziewana jest zmienna typu `MojaLiczba`, np.:

```
MojaLiczba x1 = 10;
```

Konwersję powyższą możemy również wywołać w sposób jawny:

```
MojaLiczba x2 = (MojaLiczba) 10;
```

Druga metoda definiuje konwersję jawną z typu `string` na typ `MojaLiczba`. W programie, aby zamienić zmienną lub literał typu `string` na typ `MojaLiczba`, musimy jawnie użyć operatora konwersji, np.:

```
MojaLiczba x3 = (MojaLiczba) "10";
```

Natomiast poniższa linijka spowoduje błąd kompilacji:

```
MojaLiczba x3 = "10";
```

Przykładowe rozwiązanie

Przeciążenie operatorów na przykładzie liczb zespolonych

W przykładzie spróbujemy pokazać przeciążenie operatorów w zastosowaniu do liczb zespolonych.

Utworzenie struktury reprezentującej liczbę zespoloną

Przypomnijmy, że liczba zespolona składa się z dwóch części: części rzeczywistej i części urojonej. Zarówno część rzeczywista, jak i część urojona, są liczbami rzeczywistymi. Liczbę zespoloną zaimplementujemy jako strukturę (nie zajmuje dużo miejsca, nie musi być zarządzana przez garbage collector, może być przechowywana na stosie). Dodajmy też odpowiednie konstruktory, pamiętając, że dla struktur nie można definiować konstruktora bezparametrowego. Przykładowa implementacja jest zamieszczona poniżej:

```
public struct Zespolona {
    public double Re { set; get; }
    public double Im { set; get; }

    public Zespolona(double re, double im) : this() {
        Re = re;
        Im = im;
    }

    public Zespolona(double re) : this(re, 0)
    { }
}
```

Przeciążenie operatorów unarnych

Chcemy posługiwać się liczbami zespolonymi w podobny sposób jak innymi typami liczbowymi. Zauważmy, że dla wszystkich typów liczbowych mamy zdefiniowane jednoargumentowe operatory + i -. Pierwszy zwraca tę samą liczbę, drugi zaś zwraca liczbę przeciwną. Dla liczb zespolonych nie zmienimy znaczenia tych operatorów i zaimplementujemy je w następujący sposób:

```
public static Zespolona operator +(Zespolona x) {
    return x;
}

public static Zespolona operator -(Zespolona x) {
    return new Zespolona(-x.Re, -x.Im);
}
```

Z każdą liczbą zespoloną jest związana liczba sprzężona. Liczba sprzężona do danej liczby różni się znakiem części urojonej. Części urojone tych liczb są liczbami przeciwnymi. Do reprezentacji liczby sprzężonej użyjemy operatora ~:

```
public static Zespolona operator ~(Zespolona x) {
    return new Zespolona(x.Re, -x.Im);
}
```

Kolejnym pojęciem związanym z liczbami zespolonymi, które możemy zaimplementować przy pomocy operatora unarnego, jest moduł liczby zespolonej. Obliczamy go jako pierwiastek kwadratowy z sumy kwadratów części rzeczywistej i części urojonej danej liczby zespolonej. Do reprezentacji modułu liczby zespolonej użyjemy operatora !:

```
public static double operator !(Zespolona x) {
    return Math.Sqrt(x.Re * x.Re + x.Im * x.Im);
}
```

Przeciążenie operatorów binarnych

Liczby zespolone, podobnie jak liczby rzeczywiste, możemy dodawać, odejmować, mnożyć i dzielić. Zdefiniujmy więc te cztery podstawowe operacje dla naszego typu. Operatory, które przeciążymy, aby zaimplementować poszczególne operacje, to: +, -, * i /. Oczywiście możemy zmienić sens poszczególnych operatorów, ale starajmy się, aby obsługa naszych klas była jak najbardziej intuicyjna. Oto implementacja poszczególnych operatorów:

```
public static Zespolona operator +(Zespolona x, Zespolona y) {
    return new Zespolona(x.Re + y.Re, x.Im + y.Im);
}

public static Zespolona operator -(Zespolona x, Zespolona y) {
    return new Zespolona(x.Re - y.Re, x.Im - y.Im);
}

public static Zespolona operator *(Zespolona x, Zespolona y) {
    return new Zespolona(x.Re * y.Re, x.Im * y.Im);
}

public static Zespolona operator /(Zespolona x, Zespolona y) {
    Zespolona z = new Zespolona();
    z.Re = (x.Re * y.Re + x.Im * y.Im) / (y.Re * y.Re + y.Im*y.Im);
    z.Im = (x.Im * y.Re - x.Re * y.Im) / (y.Re * y.Re + y.Im * y.Im);
    return z;
}
```

Przeciążenie operatorów relacji

Dwie liczby zespolone są sobie równe, kiedy część rzeczywista pierwszej liczby jest równa części rzeczywistej drugiej liczby i kiedy część urojona pierwszej liczby jest równa części urojonej drugiej liczby. Opierając się na powyższej definicji możemy łatwo zdefiniować operator równości. Pamiętajmy, że operatory relacji definiujemy parami. Przeciążając operator ==, musimy również zdefiniować operator !=:

```
public static bool operator ==(Zespolona x, Zespolona y) {
    return x.Re == y.Re && x.Im == y.Im;
}

public static bool operator !=(Zespolona x, Zespolona y) {
    return ! (x == y);
}
```

Przeciążając operator ==, jak to już było napisane wcześniej, powinniśmy nadpisać metody typu Object: Equals i GetHashCode. Szczegóły na temat nadpisywania metod typu object można znaleźć w module 8 tego kursu. Tutaj podamy tylko przykładową implementację tych metod, bez zagłębiania się w szczegóły:

```
public override bool Equals(object obj) {
    if(obj == null || !(obj is Zespolona) )
        return false;
    Zespolona y = (Zespolona) obj;
    return this.re == y.re && this.im == y.im;
}

public override int GetHashCode() {
    return (int) !this;
}
```

Przeciążenie operatorów konwersji

Każda liczba rzeczywista może być traktowana jako liczba zespolona, której część urojona ma wartość równą zero. Zamiana liczby rzeczywistej na zespoloną nie grozi żadną utratą informacji, więc operator konwersji dokonujący powyższej zamiany może być niejawny. Poniżej umieszczono przykładową implementację tego operatora:

```
public static implicit operator Zespolona(double d) {  
    Zespolona x = new Zespolona();  
    x.re = d;  
    return x;  
}
```

Liczbę rzeczywistą z liczby zespolonej możemy otrzymać przez odrzucenie części urojonej. Przekształcenie to jest jednak związane z utratą informacji, dlatego operator konwersji dokonujący powyższej zamiany powinien być jawny:

```
public static explicit operator double(Zespolona x) {  
    return x.Re;  
}
```

Gotowe rozwiązanie powyższego przykładu znajduje się w katalogu **Demo\Modul05**.

Porady praktyczne

- Nie wszystkie operatory w języku C# można przeciążyć. Nie możesz przeciążyć np. operatorów: =, (), ->.
- Nie możesz tworzyć nowych operatorów np. operatora <>.
- Nie możesz zmienić priorytetu ani łączności operatorów. Operator mnożenia * zawsze będzie miał wyższy priorytet niż operator dodawania +. Operator odejmowania - zawsze będzie lewostronnie łączny.
- Nie możesz zmienić liczby argumentów funkcji operatorowej. Operator dzielenia zawsze będzie operatorem dwuargumentowym.
- Przeciążając operatory nie zmieniaj ich znaczenia. To, co na początku wydaje się dowcipne i śmieszne, może powodować w przyszłości liczne kłopoty. Sam również możesz zapomnieć, że w danej klasie zmieniłeś znaczenie danego operatora.
- Zanim zastosujesz mechanizm przeciążenia operatorów, zastanów się dobrze, czy nie lepiej użyć zwykłej metody. W przypadku, gdy dana funkcjonalność w żaden sposób nie jest związana z operatorem, zaimplementowanie jej przy pomocy mechanizmu przeciążenia operatorów nie ułatwi korzystania z danej klasy, a może wręcz to skomplikować. Zwykłej metodzie zawsze możesz nadać dowolną nazwę, informującą co metoda robi. Metodzie reprezentującej operator nazwy nadać nie możesz.
- Unikaj przeciążenia operatorów bitowych (<<, >>, &, |, ~, ^,) dla typów referencyjnych.
- Nie modyfikuj argumentów przesyłanych do metody implementującej operator. Przyjęte jest że operatory nie zmieniają wartości swoich argumentów.
- Operatory konwersji definiuj bardzo rozważnie. Nie definiuj operatora konwersji na siłę – nie każda konwersja ma sens.
- Definiuj konwersje niejawną bardzo ostrożnie. Operator konwersji niejawnej z typu A na typ B, wyłącza możliwości kontrolne kompilatora przy przepisywaniu wartości zmiennej typu A do zmiennej typu B np. przy przesyłaniu argumentów do metody.
- Konwersję niejawną z typu A na typ B powinieneś definiować tylko wtedy, gdy potrafimy zamienić obiekt klasy A na obiekt klasy B bez utraty informacji. W przypadku, gdy istnieje możliwość transformacji obiekt klasy A na obiekt klasy B, jednak zachodzi ryzyko utraty części informacji, powinieneś zawsze definiować jawny operator konwersji.
- Pamiętaj, że przeciążając operatory == oraz != powinieneś również nadpisać metodę Equals. Szczegóły na temat tej metody znajdują się w module 8.

- Zastanów się, czy zamiast przeciążać operatory relacji, nie lepiej jest zaimplementować interfejs `IComparable` lub `IComparer`. Zastosowanie interfejsów jest sposobem bardziej uniwersalnym, elastycznym i powszechniej używanym w bibliotece .NET Framework. Metody interfejsu mogą być zaimplementowane jako metody wirtualne, funkcje operatorowe niestety nie. Interfejsy zostaną omówione w modułach 9 i 11 tego kursu, a metody wirtualne w module 8.
- Operatory w języku pośrednim (IL) są reprezentowane przez ukryte metody. Standard ECMA zawiera listę nazw metod odpowiadających poszczególnym operatorom np.: operatorowi dodawania odpowiada metoda `op_Addition`, a operatorowi mniejszości metoda `op_LessThan`. Umożliwia to programistom korzystającym z języków, które nie wspierają mechanizmu przeciążenia operatorów, czy samych operatorów, na skorzystanie z funkcjonalności zdefiniowanej przy pomocy operatorów, przez wywołanie odpowiedniej metody.
- Pamiętaj, w odróżnieniu np. od języka C++, konstruktor z jednym parametrem nie definiuje konwersji z typu parametru na typ w którym konstruktor jest zdefiniowany.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- znasz i rozumiesz pojęcie przeciążenia operatorów
- wiesz, które operatory można przeciążyć w języku C#
- znasz operatory, których przeciążenie uzyskujemy nie wprost, ale przez definicję innych operatorów
- znasz wymagania obowiązujące metodę implementującą operator
- potrafisz przeciążyć poszczególne operatory
- wiesz, które operatory należy przeciążać parami
- wiesz, na czym polega konwersja
- rozumiesz różnice między konwersją jawną i niejawną
- potrafisz definiować operację konwersji między różnymi typami
- wiesz kiedy definiować konwersję jawną, a kiedy niejawną

Dodatkowe źródła informacji

1. Daniel Solis, *Illustrated C# 2008*, Apress, 2008

Książka dla tych wszystkich którzy pragną nauczyć się tworzyć programy w języku C#. Zawiera dokładne omówienie tego języka.

2. Andrew Troelsen, *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition*, Apress, 2007

Książka przeznaczona jest dla bardziej zaawansowanych programistów. Czwarte wydanie tej książki opisuje język C# 3.0 i platformę .NET 3.5.

3. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka skierowana do programistów chcących nauczyć się programować w języku C#.

4. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad, jak powinniśmy pisać swoje programy.

5. Codeguru, <http://www.codeguru.pl>

Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.

6. Kurs C#, cz. I, http://www.centrumxp.pl/dotNet/20,1,kategoria,Kurs_C_cz_I.aspx

Przystępny kurs języka C# w języku polskim.

Laboratorium podstawowe

Problem 1 (czas realizacji 25 minut)

Pracujesz w instytucie obliczeniowym. Do instytutu zgłosiła się pewna firma z prośbą o napisanie programu symulującego wybrane zjawiska fizyczne. Jednym z wymagań stawianym przez zamawiającą firmę jest, aby program pracował na platformie .NET. Instytut do tej pory nie miał doświadczenia z tym środowiskiem, nie posiada odpowiednich bibliotek numerycznych, a do symulacji części zjawisk konieczne są macierze. Zarząd instytutu podjął decyzję o stworzeniu biblioteki implementującej macierze od podstaw. Niestety to ty jesteś tym szczęśliwcem, który ma zaimplementować żadaną bibliotekę. Operacje, które musisz zaimplementować to:

- dodawanie i odejmowanie dwóch macierzy
- mnożenie dwóch macierzy
- mnożenie macierzy przez liczbę rzeczywistą
- zamiana liczby rzeczywistej na macierz o wymiarach jeden na jeden
- odwołanie się do elementu macierzy przy pomocy dwóch indeksów

Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition typu Class Library	<ul style="list-style-type: none"> • Otwórz Visual C# 2008 Express Edition. • Z menu wybierz File -> New Project. • Z listy Visual Studio installed templates wybierz Class Library. • W polu Name wpisz Macierz. • Kliknij OK. • Z menu wybierz File -> Save Macierz. • W polu Location wybierz folder w którym będzie zapisany projekt. • Zaznacz pole wyboru Create directory for solution. • W polu Solution Name wpisz Modul05. • Naciśnij przycisk Save.
2. Zaimplementuj szkielet klasy Macierz	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na Macierz.cs. W okienku dialogowym naciśnij przycisk Tak. • Do klasy Macierz dodaj pole typu tablica dwuwymiarowa liczb double, w której będą przechowywane elementy macierzy: <pre>private double[,] macierz;</pre> • Do klasy Macierz dodaj konstruktor. Do konstruktora przekaz liczbę wierszy i kolumn tworzonej macierzy: <pre>public Macierz(int liczbaWierszy, int liczbaKolumn) { macierz = new double[liczbaWierszy, liczbaKolumn]; }</pre> • Do klasy Macierz dodaj indeksator, przy pomocy którego będzie można uzyskać dostęp do poszczególnych elementów macierzy: <pre>public double this[int wiersz, int kolumna] {</pre>

	<pre> set { macierz[wiersz-1, kolumna-1] = value; } get { return macierz[wiersz-1, kolumna-1]; } } </pre> <ul style="list-style-type: none"> Do klasy Macierz dodaj właściwości, przy pomocy których będzie można otrzymać liczbę wierszy oraz liczbę kolumn macierzy: <pre> public int LiczbaWierszy { get { return macierz.GetLength(0); } } public int LiczbaKolumn { get { return macierz.GetLength(1); } } </pre>
3. Do klasy Macierz dodaj metody definiujące żądane operatory	<ul style="list-style-type: none"> Do klasy Macierz dodaj metodę implementującą dodawanie dwóch macierzy: <pre> public static Macierz operator +(Macierz a, Macierz b) { if(a.LiczbaWierszy != b.LiczbaWierszy a.LiczbaKolumn != b.LiczbaKolumn) { throw new ArgumentException("Zły rozmiar macierzy"); } Macierz c = new Macierz(a.LiczbaWierszy, a.LiczbaKolumn); for (int i = 0; i < a.LiczbaWierszy; i++) { for (int j = 0; j < a.LiczbaKolumn; j++) { c.macierz[i, j] = a.macierz[i, j] + b.macierz[i, j]; } } return c; } </pre> Do klasy Macierz dodaj metodę implementującą odejmowanie dwóch macierzy: <pre> public static Macierz operator -(Macierz a, Macierz b) { if (a.LiczbaWierszy != b.LiczbaWierszy a.LiczbaKolumn != b.LiczbaKolumn) { throw new ArgumentException("Zły rozmiar macierzy"); } Macierz c = new Macierz(a.LiczbaWierszy, a.LiczbaKolumn); for (int i = 0; i < a.LiczbaWierszy; i++) { for (int j = 0; j < a.LiczbaKolumn; j++) { c.macierz[i, j] = a.macierz[i, j] - b.macierz[i, j]; } } return c; } </pre> Do klasy Macierz dodaj metodę implementującą mnożenie dwóch macierzy: <pre> public static Macierz operator *(Macierz a, Macierz b) { if (a.LiczbaKolumn != b.LiczbaWierszy) { throw new ArgumentException("Zły rozmiar macierzy"); } Macierz c = new Macierz(a.LiczbaWierszy, b.LiczbaKolumn); for (int i = 0; i < a.LiczbaWierszy; i++) { for (int j = 0; j < b.LiczbaKolumn; j++) { c.macierz[i, j] = 0; for (int k = 0; k < b.LiczbaWierszy; k++) { c.macierz[i, j] += a.macierz[i, k] * b.macierz[k, j]; } } } return c; } </pre>

	<pre> } </pre> <ul style="list-style-type: none"> Do klasy Macierz dodaj metodę implementującą mnożenie macierzy przez liczbę rzeczywistą: <pre> public static Macierz operator *(double x, Macierz a) { Macierz c = new Macierz(a.LiczbaWierszy, a.LiczbaKolumn); for (int i = 0; i < a.LiczbaWierszy; i++) { for (int j = 0; j < a.LiczbaKolumn; j++) { c.macierz[i, j] = x * a.macierz[i, j]; } } return c; } </pre> Do klasy Macierz dodaj metodę zamieniającą liczbę rzeczywistą w macierz o wymiarach jeden na jeden: <pre> public static explicit operator Macierz(double x) { Macierz c = new Macierz(1, 1); c.macierz[0, 0] = x; return c; } </pre>
4. Do bieżącego rozwiązania dodaj nowy projekt, w którym przetestujesz nowo utworzoną klasę Macierz	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestMacierz. Kliknij OK.
5. Zaznacz projekt TestMacierz jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestMacierz, a następnie z menu kontekstowego wybierz Set as StartUp Project.
6. Do projektu TestMacierz dodaj odwołanie do biblioteki Macierze	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projektu TestMacierz, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt Macierze, a następnie kliknij przycisk OK.
7. W projekcie TestMacierz napisz kod testujący klasę Macierz	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Na górze pliku Program.cs zaimportuj przestrzeń nazw Macierze. <pre> using Macierze; </pre> Wewnątrz przestrzeni nazw TestMacierz napisz klasę pomocniczą, która będzie odpowiedzialna za wypisanie macierzy na ekranie i ich inicjalizację: <pre> static class MacierzDlaKonsoli { public static void WypiszMacierz(this Macierz a) { Console.WriteLine("["); for (int i = 1; i <= a.LiczbaWierszy; i++) { Console.WriteLine("<"); for (int j = 1; j <= a.LiczbaKolumn; j++) { Console.WriteLine(a[i, j]); if (j != a.LiczbaKolumn) Console.WriteLine("; "); } Console.WriteLine("> "); } } } </pre>

	<pre> Console.WriteLine(""); } private static Random generator = new Random(); public static void InicjalizujMacierz(this Macierz a) { for (int i = 1; i <= a.LiczbaWierszy; i++) { for (int j = 1; j <= a.LiczbaKolumn; j++) { a[i,j] = generator.Next(0,11); } } } } </pre> <ul style="list-style-type: none"> Do metody Main dodaj kod, który przetestuje działanie klasy Macierz: <pre> Macierz a = new Macierz(3, 2), b = new Macierz(3, 2); a.InicjalizujMacierz(); Console.WriteLine("Macierz A: "); a.WypiszMacierz(); Console.WriteLine("\nMacierz B: "); b.InicjalizujMacierz(); b.WypiszMacierz(); Console.WriteLine("\nMacierz A + B: "); (a + b).WypiszMacierz(); Console.WriteLine("\nMacierz A - B: "); (a - b).WypiszMacierz(); Console.WriteLine("\nMacierz 2 * A: "); (2*a).WypiszMacierz(); Macierz c = new Macierz(2, 1); c.InicjalizujMacierz(); Console.WriteLine("\nMacierz C: "); c.WypiszMacierz(); Console.WriteLine("\nMacierz A * C: "); (a * c).WypiszMacierz(); Macierz d = (Macierz)33; Console.WriteLine("\nMacierz D: "); d.WypiszMacierz(); Console.ReadKey(); </pre>
8. Skompiluj i uruchom program	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Problem 2 (czas realizacji 20 minut)

W wielu programach w firmie w której pracujesz, korzysta się z kolekcji napisów. Kolekcje trzeba łączyć, sprawdzić czy zawierają wspólne elementy itp. Postanowiono więc zaimplementować klasę reprezentującą zbiór napisów. W kasie reprezentującej zbiór trzeba zaimplementować następujące operacje:

- dodanie elementu do zbioru
- sprawdzenie czy element należy do zbioru
- dostęp do elementu zbioru przy pomocy indeksu
- pobranie liczby elementów zbioru (moc zbioru)
- utworzenie sumy dwóch zbiorów
- utworzenie różnicy dwóch zbiorów
- utworzenie iloczynu (części wspólnej) dwóch zbiorów

Wskazówka: Przy implementacji klasy reprezentującej zbiór możesz wykorzystać klasę **List<string>** do przechowywania elementów zbioru.

Zadanie	Tok postępowania
1. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. Z listy Visual Studio installed templates wybierz Class Library. W polu Name wpisz Zbiory. Kliknij OK.
2. Zaimplementuj szkieleł klasy ZbiorNapisow	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na ZbiorNapisow.cs. W okienku dialogowym naciśnij przycisk Tak. Do klasy ZbiorNapisow dodaj pole, które będzie przechowywać elementy zbioru: <pre>private List<string> napisy = new List<string>();</pre> Do klasy ZbiorNapisow dodaj metodę, przy pomocy której będzie można dodać element do zbioru: <pre>public void DodajElement(string s) { if (napisy.Contains(s)) napisy.Add(s); }</pre> Do klasy ZbiorNapisow dodaj metodę, przy pomocy której będzie można sprawdzić, czy zbiór zawiera już dany element : <pre>public bool CzyElementNalezy(string element) { return napisy.Contains(element); }</pre> Do klasy ZbiorNapisow dodaj indeksator, przy pomocy którego będzie można uzyskać dostęp do poszczególnych elementów zbioru: <pre>public string this[int indeks] { get { return napisy[indeks]; } }</pre> Do klasy ZbiorNapisow dodaj właściwości, przy pomocy których będzie można otrzymać liczbę elementów zbioru: <pre>public int MocZbioru { get { return napisy.Count; } }</pre>
3. Do klasy ZbiorNapisow dodaj metody definiujące żądane operatory	<ul style="list-style-type: none"> Do klasy ZbiorNapisow dodaj metodę implementującą dodawanie dwóch zbiorów: <pre>public static ZbiorNapisow operator +(ZbiorNapisow a, ZbiorNapisow b) { ZbiorNapisow c = new ZbiorNapisow(); foreach (string s in a.napisy) { c.DodajElement(s); } foreach (string s in b.napisy) { c.DodajElement(s); } return c; }</pre> Do klasy ZbiorNapisow dodaj metodę implementującą różnicę dwóch zbiorów:

	<pre> public static ZbiorNapisow operator -(ZbiorNapisow a, ZbiorNapisow b) { ZbiorNapisow c = new ZbiorNapisow(); foreach (string s in a.napisy) { if (! b.CzyElementNalezy(s)) c.DodajElement(s); } return c; } </pre> <ul style="list-style-type: none"> Do klasy ZbiorNapisow dodaj metodę implementującą część wspólną dwóch zbiorów: <pre> public static ZbiorNapisow operator *(ZbiorNapisow a, ZbiorNapisow b) { ZbiorNapisow c = new ZbiorNapisow(); foreach (string s in a.napisy) { if (b.CzyElementNalezy(s)) c.DodajElement(s); } return c; } </pre>
4. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestZbiory. Kliknij OK.
5. Zaznacz projekt TestZbiory jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestZbiory, a następnie z menu kontekstowego wybierz Set as StartUp Project.
6. Do projektu TestZbiory dodaj odwołanie do biblioteki Sortowanie	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projektu TestZbiory, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt Zbiory, a następnie kliknij przycisk OK.
7. Zaimplementuj metodę Main	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Na górze pliku Program.cs zaimportuj przestrzeń nazw Zbiory: <pre>using Zbiory;</pre> Do metody Main dodaj kod, który przetestuje działanie klasy ZbiorNapisow: <pre> ZbiorNapisow a = new ZbiorNapisow(); a.DodajElement("kot"); a.DodajElement("pies"); a.DodajElement("koza"); ZbiorNapisow b = new ZbiorNapisow(); b.DodajElement("krowa"); b.DodajElement("owca"); b.DodajElement("koza"); Console.WriteLine("Zbior A:={ "); for (int i = 0; i < a.MocZbioru; i++) Console.WriteLine(" {0},", a[i]); Console.WriteLine("}"); Console.WriteLine("Zbior B:={ "); for (int i = 0; i < b.MocZbioru; i++) </pre>

	<pre> Console.Write(" {0},", b[i]); Console.WriteLine(""); ZbiorNapisow c = a + b; Console.Write("Zbior A+B:={ "); for (int i = 0; i < c.MocZbioru; i++) Console.Write(" {0},", c[i]); Console.WriteLine(""); c = a - b; Console.Write("Zbior A-B:={ "); for (int i = 0; i < c.MocZbioru; i++) Console.Write(" {0},", c[i]); Console.WriteLine(""); c = a * b; Console.Write("Zbior A*B:={ "); for (int i = 0; i < c.MocZbioru; i++) Console.Write(" {0},", c[i]); Console.WriteLine(""); Console.ReadKey(); </pre>
8. Skompiluj i uruchom program	<ul style="list-style-type: none"> • Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. • W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 90 min)

Pracujesz w instytucie obliczeniowym. Do instytutu zgłosiła się pewna firma z prośbą o napisanie programu symulującego wybrane zjawiska fizyczne. Jednym z wymagań stawianym przez zamawiającą firmę jest, aby program pracował na platformie .NET i obliczenia dokonywał z dokładnością bezwzględną do 20 cyfr po przecinku. Instytut do tej pory nie miał doświadczenia z tym środowiskiem i nie posiada odpowiednich bibliotek numerycznych. Zarząd instytutu podjął decyzję o stworzenia biblioteki implementującej „duże liczby”, dla których można określić konieczną dokładność bezwzględną (do ilu miejsc po przecinku powinny być dokonywane obliczenia) i w zasadzie nie ma ograniczenia co do wielkość liczb. Niestety to ty jesteś tym szczęśliwcem, który ma zaimplementować żadaną bibliotekę. Operacje, które musisz zaimplementować, to:

- dodawanie i odejmowanie dwóch dużych liczb
- mnożenie dwóch dużych liczb
- mnożenie dużej liczby przez liczbę rzeczywistą
- zamiana liczby rzeczywistej (zarówno typu `double`, jak i `decimal`) na dużą liczbę
- zamiana łańcucha znaków na dużą liczbę
- zamiana dużej liczby na liczbę rzeczywistą (zarówno na typ `double`, jak i `decimal`)
- porównywanie dwóch dużych liczb

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 6

Wersja 2

Dziedziczenie

Spis treści

Dziedziczenie	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	7
Porady praktyczne	10
Uwagi dla studenta	11
Dodatkowe źródła informacji	11
Laboratorium podstawowe	13
Problem 1 (czas realizacji 45 minut)	13
Laboratorium rozszerzone	18
Zadanie 1 (czas realizacji 90 min)	18

Informacje o module

Opis modułu

W tym module zapoznasz się z jednym z podstawowych pojęć programowania obiektowego, jakim jest dziedziczenie. Nauczysz się, jak wykorzystywać dziedziczenie w języku C# i poznasz pozostałe modyfikatory dostępu – `protected`, `internal` oraz `internal protected`. Dowiesz się również, jak inicjalizować obiekty klas pochodnych oraz jak przesłaniać metody klasy bazowej.

Cel modułu

Celem modułu jest pokazanie, co to jest dziedziczenie i jak korzystać z tego mechanizmu w języku C#.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- znał pojęcie dziedziczenia
- potrafił definiować klasy pochodne w języku C#
- wiedział, do czego służą modyfikatory dostępu `protected`, `internal` i `internal protected`
- potrafił przesłonić metodę klasy bazowej

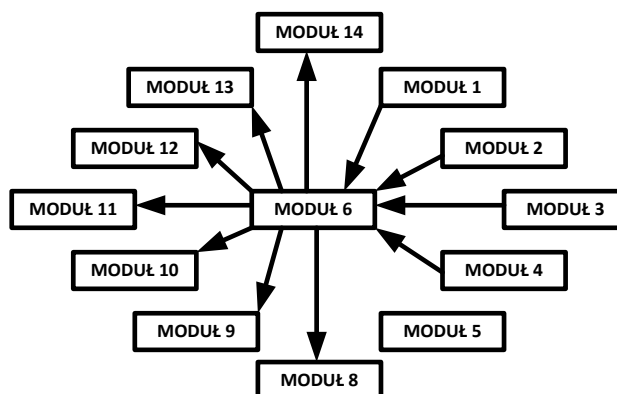
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinieneś:

- potrafić definiować klasę
- wiedzieć, co to jest pole i co to jest metoda
- znać modyfikatory dostępu `public` i `private`
- znał pojęcie konstruktora i potrafił go definiować

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w modułach „Pojęcie klasy”, „Konstruktor”, „Właściwości i indeksatory” oraz „Składowe statyczne”.



Rys. 8 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Pisząc własne programy często korzystasz z gotowych klas, no bo po co wywarzać otwarte już drzwi. Wszystko jest dobrze, dopóki dana klasa spełnia w pełni Twoje oczekiwania. Może zdarzyć się sytuacja, że klasa nie do końca zaspokaja Twoje wymagania, a jak to stwierdzono już w pewnej reklamie, „prawie robi wielką różnicę”. Oczywiście możesz spróbować napisać własną klasę od początku, ale w gotowej klasie chodzi tylko o zmianę działania jednej metody i dodanie dodatkowego pola. Implementacja nowej metody nie powinna zająć więcej niż godzinę, a utworzenie własnej klasy to zorganizowanie sobie czasu na kilka dni. Metody rozszerzające nie wchodzi w grę, bo jak już zapewne czytałeś w module 4 tego kursu, przy ich pomocy nie można dodać pola, ani też nadpisać istniejącej metody. Pozostaje więc modyfikacja kodu źródłowego gotowej klasy. Tutaj możesz napotkać jednak poważne trudności. Po pierwsze nie zawsze masz dostęp do kodu źródłowego. Po drugie modyfikacja kodu, zwłaszcza napisanego przez kogoś innego, może prowadzić do rozstrojenia klasy. Najlepszym rozwiązaniem tego problemu jest zastosowanie dziedziczenia. Jeśli chcesz wiedzieć więcej, zapraszam do dalszej lektury tego modułu.

Podstawy teoretyczne

Dziedziczenie (ang. *inheritance*) w programowaniu obiektowym to mechanizm pozwalający tworzyć nowe klasy na podstawie już istniejących. Nową klasę nazywamy *klasą pochodną*, *podklasą* lub *klasą potomną*, zaś klasę, na bazie której tworzymy klasę pochodną, nazywamy *klasą bazową*, *nadklasą* lub *klasą podstawową*. W wyniku dziedziczenia, klasa pochodna otrzymuje (dziedziczy) wszystkie pola i metody klasy bazowej.

Dziedziczenie definiuje pewien związek między klasami. Związek ten określa, że pewna klasa (klasa pochodna) jest szczególnym przypadkiem innej klasy (klasy bazowej). Dlatego też dziedziczenie nazwane jest często *konkretyzacją* lub *uszczegółowieniem*. Związek, że coś jest rodzajem czegoś występuje w życiu codziennym. Samochód czy motocykl jest pewnym rodzajem pojazdu. Podobnie student czy pracownik to uszczegółowienie pojęcia osoba. Możemy więc o studencie wypowiadać się jako o osobie, natomiast samochód traktować jako pojazd. Przenosząc to na język programowania możemy stwierdzić, że obiekt klasy Student może być używany jako obiekt klasy Osoba, czyli możemy uzyskać do niego dostęp przy pomocy zmiennej klasy Osoba. Bardziej ogólnie możemy powiedzieć, że zmienną klasy bazowej można powiązać z obiektem klasy pochodnej, co demonstruje poniższy przykład:

```
Bazowa b1 = new Pochodna();
Pochodna p1 = new Pochodna();
Bazowa b2 = p1;
```

Podsumowując, za zmienną klasy bazowej możemy zawsze podstawić zmienną klasy pochodnej. Odwrotnie, nawet gdy zmienna klasy bazowej odwołuje się do klasy pochodnej, nie można bezpośrednio przypisać jej do zmiennej klasy pochodnej. W dalszej części tego kursu zostanie opisane, jak zdefiniować takie przypisanie.

Powiedzieliśmy już dużo o samym dziedziczeniu, ale nie pokazaliśmy, jak można je zdefiniować. W języku C# nową klasę tworzymy na podstawie klasy już istniejącej w następujący sposób:

```
[modyfikator_dostepu] class nazwa_klasy_pochodnej : nazwa_klasy_bazowej
{
    // definicja dodatkowych składowych klasy pochodnej
}
```

czyli mając klasę Osoba zdefiniowaną jak poniżej:

```
class Osoba {
    public string Imie {set; get;}
```

```
        public string Nazwisko {set; get;}  
    }
```

możemy zdefiniować klasę Student, która będzie zawierała wszystkie składniki klasy Osoba, a dodatkowo np. pole numerIndeksu, pisząc:

```
class Student : Osoba {  
{  
    public int NumerIndeksu {set; get;}  
    public string InformacjeOStudencie() {  
        return string.Format("Student(ka) {0} {1} o numerze indeksu",  
            Imie, Nazwisko, NumerIndeksu);  
    }  
}
```

Obiekty klasy Student będą zawierać składowe zdefiniowane zarówno w klasie Osoba, jak i bezpośrednio w klasie Student:

```
Student s1 = new Student();  
s1.Imie = "Jan";           //odwołanie się do składowej odziedziczonej  
s1.NumerIndeksu = 100;     //odwołanie się do własnej składowej
```

Klasa bazowa wraz ze wszystkimi klasami pochodnymi i pochodnymi klas pochodnych (itd.) tworzą pewien układ hierarchiczny (klasa nadrzędna-klasa podrzędna), który nazywamy *hierarchią klas*. W języku C# wszystkie typy tworzą jedną wspólną hierarchię klas, ponieważ każdy typ dziedziczy, pośrednio lub bezpośrednio, po typie Object. Poniższa definicja klasy:

```
class K  
{ }
```

jest równoważna:

```
class K : object  
{ }
```

Typ object zostanie dokładniej opisany w module 8.

W przypadku definicji struktur nie możemy określić typu bazowego. Wszystkie struktury dziedziczą domyślnie po typie ValueType, który wywodzi się z typu object. Typ strukturalny nie może występować jako typ bazowy przy dziedziczeniu. Upraszczając możemy stwierdzić, że w języku C# mechanizm dziedziczenia nie dotyczy struktur.

Modyfikatory dostępu

Widomo, że w klasie możemy definiować składowe publiczne i prywatne. Oczywiście oba rodzaje składników są dziedziczone przez klasę pochodną. Do składowych prywatnych klasy bazowej nie możemy się jednak odwoływać z metod klasy pochodnej. Wiadomo, składnik prywatny jest dostępny tylko dla metod klasy, w której jest zdefiniowany. Składniki publiczne są dostępne dla metod wszystkich klas. W językach obiektowych istnieje jeszcze jeden modyfikator dostępu o nazwie protected. Składowe opatrzone tym modyfikatorem nazywamy *składowymi chronionymi*. Są one dostępne dla metod klasy, w której zostały zdefiniowane oraz dla metod klas pochodnych (klas dziedziczących po klasie, w której zdefiniowany jest składnik protected). W niektórych podręcznikach można znaleźć stwierdzenie, że składowe chronione w klasach pochodnych od danej klasy są traktowane, jakby były składowymi publicznymi. Nie jest to do końca prawdą, co pokażemy na następującym przykładzie.

```
class Bazowa {  
    private int x;  
    protected int y;  
    public int z;  
}
```

```

class Pochodna : Bazowa {
    public void F(Pochodna p) {
        //x = p.x;           //Błąd, nie mamy dostępu do składowych prywatnych
                             //klasy bazowej
        y = p.y;             //OK
        z = p.z;             //OK, składowe publiczne
    }

    public void G(Bazowa b) {
        //x = b.x;           //Błąd, nie mamy dostępu do składowych prywatnych
                             //klasy bazowej
        //y = b.y;           //Błąd!!! Nie możemy odwołać się do składowej
                             //chronionej przy pomocy zmiennej klasy bazowej
        z = b.z;             //OK, składowe publiczne
    }
}

```

Uściślijmy więc, składowe chronione są dostępne w klasach pochodnych pod warunkiem, że uzyskujemy do nich dostęp przy pomocy zmiennych danej klasy pochodnej.

W języku C# występują jeszcze dwa modyfikatory dostępu: `internal` oraz `internal protected`. Składowe `internal` są dostępne dla wszystkich metod należących do tego samego zestawu (ang. *assembly*). Ten modyfikator jest domyślny dla wszystkich elementów definiowanych bezpośrednio wewnątrz przestrzeni nazw. Definiując więc klasę lub strukturę bezpośrednio w przestrzeni nazw i nie określając modyfikatora dostępu powodujemy, że nie będzie jej można użyć w innych zestawach. Modyfikator `internal protected` oznacza, że dana składowa jest dostępna dla metod wszystkich klas należących do tego samego zestawu oraz dla metod klas pochodnych od danej klasy, nawet gdy nie należą do tego samego zestawu. Oczywiście dostęp z metod klas pochodnych nie należących do tego samego zestawu jest ograniczony tylko do pól danej klasy pochodnej.

```

//Zestaw pierwszy
public class K1 {
    internal int pole1;
    internal protected int pole2;
}

internal class K2 {
    public static void f(K1 k) {
        k.pole1 = 10; //OK, klasa należy do tego samego zestawu
        k.pole2 = 20; //OK, klasa należy do tego samego zestawu
    }
}

//Zestaw drugi
public class C1 {
    public static void f(K1 k) {
        //k.pole1 = 10; //Błąd, klasa K1 należy do innego zestawu
        //k.pole2 = 20; //Błąd, klasa K1 należy do innego zestawu
        //K2 k2;        //Błąd, klasa K2 należy do innego zestawu i
                        //jest "internal"
    }
}

public class C2 : K1 {
    public static void f(K1 k, C2 c) {
        //k.pole1 = 10; //Błąd, klasa K1 należy do innego zestawu
        //k.pole2 = 20; //Błąd, klasa K1 należy do innego zestawu
        //c.pole1 = 10; //Błąd, klasa K1 należy do innego zestawu,
        c.pole2 = 20;   //OK, pole protected internal
    }
}

```

Przeanalizujemy jeszcze, gdzie można użyć poszczególnych modyfikatorów dostępu. Modyfikatory, których można użyć do wszystkich elementów, to `internal` i `public`. Pozostałych modyfikatorów nie możemy stosować do elementów definiowanych bezpośrednio wewnątrz przestrzeni nazw. Z modyfikatorów `protected` i `internal protected` nie możemy również skorzystać w przypadku elementów definiowanych wewnątrz struktury, ponieważ po typie strukturalnym nie wolno dziedziczyć.

Inicjalizacja klas pochodnych

Na początku tego modułu stwierdziliśmy, że klasa pochodna otrzymuje w spadku wszystkie składowe klasy podstawowej. Jest pewien wyjątek. Dla klasy pochodnej musimy dostarczyć jej własny zestaw konstruktorów. Powód wydaje się oczywisty: po pierwsze, konstruktor posiada nazwę klasy, w której jest zdefiniowany, a po drugie, obiekty klas pochodnych, to „coś więcej” niż obiekt klasy podstawowej. To „coś więcej” w konstruktorze klasy bazowej na pewno nie zostało zainicjalizowane.

Obiekt klasy pochodnej możemy podzielić na dwie części. Pierwsza część to pola odziedziczone po klasie podstawowej. Drugą część stanowią pola dodane w klasie pochodnej. Pisząc własny konstruktor dla klasy pochodnej musimy zadbać o zainicjalizowanie tej drugiej części. Pierwsza część jest inicjalizowana przy pomocy konstruktora bezargumentowego klasy bazowej. Konstruktor bezargumentowy klasy bazowej jest wywoływany automatycznie przed konstruktorem klasy pochodnej. Jak to ładnie określił Jerzy Grębosz w książce „Symfonia C++”: „klasa najpierw uszanuje starszych, a dopiero na końcu zajmie się sama sobą”. Z kolejności wywołania konstruktorów wynikają następujące konsekwencje:

- w konstruktorze klasy pochodnej pola odziedziczone są już prawidłowo zainicjalizowane
- w konstruktorze klasy pochodnej możemy nadpisać wartości pól klasy podstawowej

Zachodzi jednak pytanie co wtedy, gdy klasa bazowa nie ma konstruktora bezparametrowego lub chcemy zainicjalizować pola klasy bazowej korzystając z innego konstruktora niż konstruktor bezparametrowy. Możemy wtedy użyć listy inicjalizacyjnej i słowa kluczowego `base`. Pokażmy to na przykładzie:

```
class Bazowa {
    public Bazowa(int i, double d)
    { }
}

class Pochodna : Bazowa {
    //public Pochodna(int i, double d) //błąd, klasa podstawowa nie ma
    //{}                               //konstruktora bezparametrowego

    public Pochodna(int i, double d) : base(i, d) //Jawne wskazanie, który
    { }                                     // konstruktor ma być wywołany
}
```

Przesłonięcie metody klasy bazowej

Wiemy już, że przy pomocy dziedziczenia możemy wzbogacić klasę bazową o nowe składowe (pola, metody, właściwości itd.), definiując odpowiednią klasę pochodną. Może się również zdarzyć, że funkcjonalność pewnej metody klasy bazowej będzie nieodpowiednia dla klasy pochodnej. Stosujemy wtedy mechanizm zwany *przesłonięciem metody klasy bazowej*, który polega na zdefiniowaniu w klasie pochodnej metody o takiej samej sygnaturze, jak metoda w klasie bazowej. Do definicji metody powinniśmy dodać modyfikator `new`. Przesłonięcie metody klasy bazowej demonstruje poniższy przykład:

```
class Bazowa {
    public void f() {
```



```
    }  
}  
  
class Pochodna : Bazowa {  
    public new void f() {  
    }  
}
```

Za pomocą tego mechanizmu możemy w klasie pochodnej zmienić dla danej metody typ wartości zwracanej oraz modyfikator dostępu.

Często przy przesłonięciu metody klasy bazowej nie chodzi nam o całkowitą zmianę funkcjonalności metody klasy bazowej, ale o dodanie kilku linijek kodu do kodu zawartego w metodzie bazowej. Zamiast przepisywać powtórnie kod metody bazowej, możemy ją wywołać stosując następującą konstrukcję:

```
base.nazwa_metody(lista_argumentow)
```

W celu wywołania przesłoniętej metody klasy bazowej, konstrukcję tę możemy stosować w dowolnej metodzie klasy pochodnej.

Przykładowe rozwiązanie

Tworzenie hierarchii klas

Załóżmy, że naszym zadaniem jest stworzenie programu ułatwiającego pracę astronomów. Program będzie służył do opisu poszczególnych ciał niebieskich.

Utworzenie klasy bazowej CialoNiebieskie

Pierwszym krokiem do utworzenia programu jest stworzenie klasy bazowej CialoNiebieskie, która zawierałaby wszystkie informacje opisujące dowolne ciało niebieskie: nazwę danego ciała, jego masę oraz opis. Pola, które będą przechowywać te informacje, uczynimy polami chronionymi (protected), aby można było mieć do nich dostęp z metod klas pochodnych. Przykładowa implementacja znajduje się poniżej:

```
public class CialoNiebieskie {  
    protected string nazwa;  
    protected double masa;  
    protected string opis;  
    public CialoNiebieskie(string nazwa,double masa,string opis) {  
        this.nazwa = nazwa;  
        this.masa = masa;  
        this.opis = opis;  
    }  
}
```

Do klasy CialoNiebieskie dodajmy dwie metody, które będą zwracać informację o danym ciele niebieskim. Pierwsza będzie zwracać informację o jego nazwie i masie, druga natomiast dodatkowo udostępniać będzie opis danego ciała niebieskiego:

```
public string ZwrocInformacje() {  
    return string.Format("Ciało niebieskie {0} ma masę {1}", nazwa, masa);  
}  
  
public string ZwrocInformacjeSzczegolowa() {  
    return string.Format("{0}\nInformacje dodatkowe:\n {1} ",  
        ZwrocInformacje() , opis );  
}
```

Utworzenie klas pochodnych reprezentujących poszczególne rodzaje ciał niebieskich

Poszczególne ciała niebieskie różnią się cechami, które je reprezentują, np. planety dodatkowo opisujemy przy pomocy takich cech jak:

- okres obiegu wokół Słońca
- czas rotacji wokół własnej osi
- ile razy dana planeta jest cięższa od Ziemi

Klasa Planeta również będzie zawierać informacje takie jak: nazwa, masa i opis. Pola te zostaną odziedziczone po klasie CialoNiebieskie. Oczywiście metody zwracające informacje na temat planety muszą również zostać zmodyfikowane. W tym celu przestonimy metody z klasy bazowej. Przykładowa implementacja klasy Planeta znajduje się poniżej:

```
public class Planeta : CialoNiebieskie {
    protected float rok;        //okres obiegu w dniach
    protected float dzien;      //okres rotacji w godz.
    public const double MASAZIEMI = 5.9736e24;

    public int IloscMasZiemi {
        get { return (int)(masa / MASAZIEMI); }
    }

    public Planeta(string nazwa, double masa, string opis,
        float rok, float dzien) : base(nazwa,masa,opis) {
        this.dzien = dzien;
        this.rok = rok;
    }

    public new string ZwrocInformacje() {
        return string.Format("Planeta {0} okrąża Słońce w {2} dni i " +
            "wykonuje obrót wokół własnej osi w {3} godz. oraz ma masę {1}",
            nazwa, masa, rok, dzien);
    }

    public new string ZwrocInformacjeSzczegolowa() {
        return string.Format("{0}\nInformacje dodatkowe:\n {1} ",
            ZwrocInformacje(), opis);
    }
}
```

Zwróćmy uwagę, że w konstruktorze klasy Planeta na liście inicjalizacyjnej musieliśmy wywołać konstruktor klasy bazowej, gdyż nie posiada ona konstruktora bezargumentowego.

Innym rodzajem ciała niebieskiego, dla którego utworzymy klasę, jest gwiazda. Gwiazdę, poza nazwą, masą i opisem, będą charakteryzować następujące cechy:

- klasa gwiazdy, oznaczona jedną z następujących liter: O, B, A, F, G, K, M, R, N, S
- podklasa gwiazdy, oznaczona cyframi od 1 do 9
- ile razy dana gwiazda jest cięższa od Słońca

Przykładowa implementacja klasy Gwiazda znajduje się poniżej:

```
public class Gwiazda : CialoNiebieskie {
    public enum KlasyGwiazd { O, B, A, F, G, K, M, R, N, S };
    protected KlasyGwiazd KlasaGwiazdy { set; get; }
    protected byte PodklasaGwiazdy { set; get; }
    public const double MASASLONCA = 1.9889e30;

    public int IloscMasSlonca {
        get { return (int)(masa / MASASLONCA); }
    }
}
```

```

    }

    public Gwiazda(string nazwa, double masa, string opis,
        KlasyGwiazd klasa, byte podklasa) : base(nazwa, masa, opis) {
        KlasaGwiazdy = klasa;
        PodklasaGwiazdy = podklasa;
    }
}

```

Podobnie jak w przypadku klasy `Planeta`, przesłoniemy metody zwracające informacje na temat ciała niebieskiego, dodając odpowiednie informacje na temat danej gwiazdy:

```

    public new string ZwrocInformacje() {
        return string.Format("Gwiazda {0} jest klasy {2}{3} i ma masę {1}",
            nazwa, masa, KlasaGwiazdy, PodklasaGwiazdy);
    }
}

```

Utworzenie własnej klasy reprezentującej wyjątek

Zauważmy, że łatwo zainicjalizować właściwość `PodklasaGwiazdy` niepoprawną wartością. Chcąc temu zapobiec, zdefiniujemy własną klasę reprezentującą wyjątek. Wyjątek będzie zgłaszany w momencie wykrycia próby nadania właściwości `PodklasaGwiazdy` złej wartości.

Klasa wyjątku w języku C# musi pośrednio lub bezpośrednio dziedziczyć po klasie `Exception`. Przykładowa definicja klasy jest zamieszczona poniżej:

```

    public class ZlaPodklasaGwiazdyException : ArgumentException {
        public ZlaPodklasaGwiazdyException()
            : base("Każda klasa ma tylko 9 podklas.") {
        }
    }
}

```

Istniejącą implementację składowej `PodklasaGwiazdy` z użyciem automatycznej właściwości zastąpimy nową, nie zapominając o dodaniu odpowiedniego pola:

```

    private byte podklasaGwiazdy;

    protected byte PodklasaGwiazdy {
        set {
            if (value < 1 || value > 9)
                throw new ZlaPodklasaGwiazdyException();
            podklasaGwiazdy = value;
        }
        get { return podklasaGwiazdy; }
    }
}

```

Utworzenie programu testującego

Przykładowy program testujący wcześniej zdefiniowane klasy jest umieszczony poniżej:

```

    static void Main(string[] args) {
        Gwiazda slonce = new Gwiazda("Słońce", 1.9889e30, "Nasz gwiazda ...",
            Gwiazda.KlasyGwiazd.G, 2);
        Console.WriteLine(slonce.ZwrocInformacje());
        Console.WriteLine(slonce.ZwrocInformacjeSzczegolowa());

        Planeta ziemia = new Planeta("Ziemia", 5.9736e24, "Nasza planeta ...",
            365.25696f, 24f);
        Console.WriteLine(ziemia.ZwrocInformacje());
        Console.WriteLine(ziemia.ZwrocInformacjeSzczegolowa());

        try {
            Gwiazda testowa = new Gwiazda("Dla testu", 2e30, "Stworzona do testu",

```

```
        Gwiazda.KlasyGwiazd.B, 12);  
    }  
    catch (Gwiazda.ZlaPodklasaGwiazdyException ex) {  
        Console.WriteLine("Uwaga wyjątek: {0}", ex.Message);  
    }  
    Console.ReadKey();  
}
```

Zwróćmy uwagę, że metoda `ZwrocInformacjeSzczegolowa` dla obiektu klasy `Gwiazda` wyświetla tylko informację zawartą w klasie `CialoNiebieskie`. Dzieje się tak, ponieważ w klasie `Gwiazda` nie przesłoniliśmy metody `ZwrocInformacjeSzczegolowa`.

Gotowe rozwiązanie powyższych przykładów znajduje się w katalogu **Demo\Modul06**.

Porady praktyczne

- Dziedziczenia nie należy stosować w przypadku, gdy klasy mają podobną implementację, lecz wtedy, gdy pojęcie modelowane przez jedną klasę jest szczególnym rodzajem pojęcia modelowanego przez drugą klasę.
- Związków typu, że coś się składa z czegoś, nie implementujemy za pomocą dziedziczenia. Do tego używamy agregacji (zawierania). Wewnątrz jednej klasy definiujemy pole, którego typ określony jest przez drugą klasę.
- Przy tworzeniu własnej hierarchii klas staraj się, żeby nie była zbyt głęboka. W większości przypadków trzy pokolenia w zupełności wystarczą.
- Pamiętaj, że wszystkie typy w języku C#, również typy bezpośrednie, w tym typy wbudowane, dziedziczą po typie `object`, pośrednio lub bezpośrednio.
- W języku C# klasa może dziedziczyć bezpośrednio tylko po jednej klasie.
- Dla struktur nie możemy określić typu bazowego. Struktura nie może być też typem bazowym.
- Definicja struktury nie może zawierać składowych `protected` i `internal protected`.
- Elementy (typy) definiowane bezpośrednio w przestrzeni nazw mogą być opatrzone tylko modyfikatorem dostępu `internal` lub `public`.
- Domyślnym poziomem dostępu (gdy pominiemy modyfikator dostępu) składowych definiowanych bezpośrednio w przestrzeni nazw jest `internal`, a dla składowych definiowanych wewnątrz klasy lub struktury `private`.
- W przypadku definicji składowych prywatnych zalecane jest określanie modyfikatora dostępu w sposób jawny.
- Pamiętaj, że przy tworzeniu obiektu klasy pochodnej najpierw wywoływany jest konstruktor klasy bazowej, a dopiero następnie wywoływany jest kod zawarty w ciele konstruktora klasy pochodnej.
- Klasa, która reprezentuje wyjątek, w języku C# musi dziedziczyć pośrednio lub bezpośrednio po klasie `Exception`.
- Tworząc nową klasę, która ma reprezentować wyjątek, dodaj do jej nazwy sufiks `Exception`.
- Tworząc klasę reprezentującą wyjątek, zapewnij że w konstruktorze tej klasy nie zostanie zgłoszony żaden wyjątek.
- Tworząc własną klasę wyjątku dodaj do niej trzy konstruktory:
 - konstruktor bezparametrowy
 - konstruktor przyjmujący pojedynczy parametr typu `string` reprezentujący komunikat
 - konstruktor przyjmujący dwa parametry. Poza komunikatem, mamy parametr reprezentujący wewnętrzny wyjątek - wyjątek który spowodował rzucenie danego wyjątku. Konstruktor ten jest używany w bloku `catch` do ponownego rzucenia wyjątku.

- Do określenia, który konstruktor klasy bazowej ma być wywoływany do zainicjalizowania części odziedziczonej, używamy słowa kluczowego `base`. Wywołanie konstruktora klasy bazowej musi być umieszczone na liście inicjalizacyjnej.
- Do wywołania przesłoniętej metody klasy bazowej również używamy słowa kluczowego `base`.
- Ponieważ klasa pochodna jest szczególnym przypadkiem klasy bazowej, zmienna klasy bazowej może zawierać referencję do obiektu klasy pochodnej.
- W przypadku, gdy w klasie pochodnej przesłoniemy metodę klasy bazowej, a zmienna klasy bazowej będzie zawierać referencję do obiektu klasy pochodnej, to gdy wywołamy przesłoniętą metodę, zostanie wywołana metoda z klasy bazowej. Dla metod przesłoniętych kompilator decyduje, którą metodę wywołać, na podstawie typu zmiennej, a nie typu obiektu.
- Pamiętaj, metody klas pochodnych mają dostęp do składowych chronionych (`protected`) klasy bazowej tylko przy pomocy zmiennych danej klasy pochodnej. Nie możesz się do nich odwołać z metod klas pochodnych przy pomocy zmiennej typu określonego przez klasę bazową.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- wiesz do czego służy i na czym polega dziedziczenie
- rozumiesz takie pojęcia, jak klasa bazowa (nadklasa, klasa podstawowa), klasa pochodna (podklasa, klasa potomna) oraz uogólnienie i uszczegółowienie
- znasz zastosowanie modyfikatorów `protected`, `internal` i `protected internal`
- wiesz jak inicjalizowany jest obiekt klasy pochodnej
- potrafisz przesłonić metodę klasy bazowej
- rozumiesz przeznaczenie słowa kluczowego `base`, zarówno w przypadku konstruktora, jak i przy przesłanianiu metod
- wiesz jak zdefiniować nową klasę reprezentującą wyjątek

Dodatkowe źródła informacji

1. Daniel Solis, *Illustrated C# 2008*, Apress, 2008

Książka dla tych wszystkich którzy pragną nauczyć się tworzyć programy w języku C#. Zawiera dokładne omówienie tego języka.

2. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka dla programistów chcących nauczyć się programować w języku C#.

3. Stephen C. Perry, *C# i .NET*, Helion, 2006

Książka w porównaniu z poprzednimi kierowana jest do trochę bardziej zaawansowanych programistów. Opisuje C# i .NET Framework w wersji 2.0

4. Andrew Troelsen, *Pro C# 2008 and the .NET 3.5 Platform*, Fourth Edition, Apress, 2007

Książka przeznaczona jest dla bardziej zaawansowanych programistów. Czwarte wydanie tej książki opisuje język C# 3.0 i platformę .NET 3.5.

5. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad jak powinniśmy pisać swoje programy.

6. Codeguru, <http://www.codeguru.pl/>

Portal polskiej społeczności programistów .Net. Nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.

7. *C Sharp Tutorial*, http://www.meshplex.org/wiki/C_Sharp_Tutorial

Internetowy kurs języka C#.

8. *C# Corner*, <http://www.csharpcorner.com>

Portal poświęcony programowaniu w języku C#.

9. *Kurs C#, cz. I*, http://www.centrumxp.pl/dotNet/20,1,kategoria,Kurs_C_cz_I.aspx

Przystępny kurs języka C# w języku polskim.

Laboratorium podstawowe

Problem 1 (czas realizacji 45 minut)




Twoim zadaniem jest zaimplementowanie i przetestowanie hierarchii składającej się z następujących klas: *Osoba*, *Student*, *Wykładowca* oraz *Stypendysta*. Poszczególne klasy zawierają następujące cechy:



- *Osoba* – imię, nazwisko, rok urodzenia oraz płeć
- *Student* – zawiera wszystkie cechy zawarte w klasie *Osoba* oraz numer indeksu
- *Wykładowca* – zawiera wszystkie cechy zawarte w klasie *Osoba* oraz tytuł (tytuł naukowy lub stopień naukowy)
- *Stypendysta* – zawiera wszystkie cechy zawarte w klasie *Student* oraz kwotę stypendium

Każda klasa powinna również zawierać:


- zestaw odpowiednich konstruktorów
- zestaw właściwości umożliwiające zmianę i odczyt odpowiednich pól
- metody zwracające w postaci napisu wszystkie informacje przechowywane w danej klasie

Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition	<ul style="list-style-type: none"> • Otwórz Visual C# 2008 Express Edition. • Z menu wybierz File -> New Project. • Z listy Visual Studio installed templates wybierz Class Library. • W polu Name wpisz Osoby. • Kliknij OK. • Z menu wybierz File -> Save Osoby. • W polu Location wybierz folder w którym będzie zapisany projekt. • Zaznacz pole wyboru Create directory for solution. • W polu Solution Name wpisz Modul06. • Naciśnij przycisk Save
2. Zaimplementuj szkielet klasy <i>Osoba</i>	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na Osoba.cs. W okienku dialogowym naciśnij przycisk Tak. • W pliku Osoba.cs bezpośrednio w przestrzeni nazw Osoby dodaj typ wyliczeniowy, który będzie służył do wyznaczenia płci: <pre>public enum Plec { Kobieta, Mezczyzna }</pre> • Do klasy Osoba dodaj żądane pola wraz z właściwościami, przy pomocy których będziemy mogli uzyskać dostęp do tych pól. Pole reprezentujące płeć uczynić dostępnym tylko z metod klasy Osoba i klas z niej dziedziczących : <pre>public class Osoba { public string Imie { set; get; } public string Nazwisko { set; get; } protected Plec Plec { set; get; } private readonly int rokUrodzenia; public int RokUrodzenia { get { return rokUrodzenia; } } }</pre> • Do klasy Osoba dodaj konstruktor, który zainicjalizuje wszystkie pola:

	<pre>public Osoba(string imie, string nazwisko, int rokUrodzenia, Plec plec) { Imie = imie; Nazwisko = nazwisko; this.rokUrodzenia = rokUrodzenia; Plec = plec; }</pre> <ul style="list-style-type: none"> Do klasy Osoba dodaj metodę zwracającą w postaci napisu pełną informację na temat obiektu klasy Osoba: <pre>public string ZwrocInformacje() { if (Plec == Plec.Kobieta) return string.Format("Pani {0} {1} urodzona w {2}", Imie, Nazwisko, RokUrodzenia); return string.Format("Pan {0} {1} urodzony w {2}", Imie, Nazwisko, RokUrodzenia); }</pre>
<p>3. Do bieżącego projektu dodaj klasę Student</p>	<ul style="list-style-type: none"> Z menu wybierz Project -> Add Class. W oknie dialogowym Add New Item – Osoby z listy Visual Studio installed templates wybierz Class. W polu Name wpisz Student. Kliknij OK. Klasę Student uczyni publiczną i zaznacz, że dziedziczy po klasie Osoba: <pre>public class Student : Osoba{ }</pre>  Jaki modyfikator byłby zastosowany do klasy Student, gdybyśmy nie poprzedzili jej definicji słowem public? Do klasy Student dodaj żądane pole i właściwość, przy pomocy której możemy uzyskać do niego dostęp: <pre>private readonly int numerIndeksu; public int NumerIndeksu { get { return numerIndeksu; } }</pre> Do klasy Student dodaj publiczny konstruktor inicjalizujący jej wszystkie pola: <pre>public Student(string imie, string nazwisko, int rokUrodzenia, Plec plec, int numerIndeksu) : base(imie, nazwisko, rokUrodzenia, plec) { this.numerIndeksu = numerIndeksu; }</pre>  Co by się stało, gdybyś na liście inicjalizacyjnej konstruktora nie wywołał w sposób jawny konstruktora klasy bazowej? Wyjaśnij dlaczego. Do klasy Student dodaj publiczny konstruktor inicjalizujący obiekt klasy Student na podstawie obiektu klasy Osoba: <pre>public Student(Osoba osoba) : base(osoba.Imie, osoba.Nazwisko, osoba.RokUrodzenia, osoba.Plec) { }</pre>  Spróbuj skompilować program. Jaki błąd zgłasza kompilator. Dlaczego nie możesz odwołać się do właściwości Plec klasy Osoba? Zmień poprzednio zdefiniowany konstruktor tak, aby inicjalizował nowy obiekt klasy Student na podstawie innego obiektu klasy Student. Po

	<p>modyfikacji kod powinien wyglądać następująco:</p> <pre>public Student(Student student) : base(student.Imie, student.Nazwisko, student.RokUrodzenia, student.Plec) { numerIndeksu = student.NumerIndeksu; }</pre> <p> Spróbuj skompilować program. Dlaczego teraz możesz się odwołać do właściwości Plec klasy Student?</p> <ul style="list-style-type: none"> Do klasy Student dodaj metodę zwracającą w postaci napisu pełną informację na temat obiektu klasy Student: <pre>public new string ZwrocInformacje() { return string.Format("{0} numer indeksu {1}", base.ZwrocInformacje(), NumerIndeksu); }</pre>
4. Do bieżącego projektu dodaj klasę Wykładowca	<ul style="list-style-type: none"> Z menu wybierz Project -> Add Class. W oknie dialogowym Add New Item – Osoby z listy Visual Studio installed templates wybierz Class. W polu Name wpisz Wykładowca. Kliknij OK. Klasę Wykładowca uczyni publiczną i zaznacz, że dziedziczy po klasie Osoba: <pre>public class Wykładowca : Osoba{ }</pre> <ul style="list-style-type: none"> Wewnątrz klasy Wykładowca zdefiniuj nowy typ wyliczeniowy, który będzie reprezentował poszczególne tytuły i stopnie naukowe: <pre>public enum Tytuły { dr, dr_hab, prof }</pre> <p> Jaki modyfikator byłby zastosowany do typu Tytuły, gdybyśmy nie poprzedzili jej definicji słowem public?</p> <ul style="list-style-type: none"> Do klasy Wykładowca dodaj żądane pole i właściwość, przy pomocy której możemy uzyskać do niego dostęp. <pre>public Tytuły Tytul { set; get; }</pre> <ul style="list-style-type: none"> Do klasy Wykładowca dodaj publiczny konstruktor inicjalizujący jej wszystkie pola: <pre>public Wykładowca(string imie, string nazwisko, int rokUrodzenia, Plec plec, Tytuły tytul) : base(imie, nazwisko, rokUrodzenia, plec) { Tytul = tytul; }</pre> <ul style="list-style-type: none"> Do klasy Wykładowca dodaj metodę zwracającą w postaci napisu pełną informację na temat obiektu klasy Student: <pre>private string zwrocTytul() { string tytul = ""; switch (Tytul) { case Tytuły.dr: tytul = "dr"; break; case Tytuły.dr_hab: tytul = "dr hab."; break; case Tytuły.prof: tytul = "prof."; break; } return tytul; }</pre> <pre>public new string ZwrocInformacje() { if (Plec == Plec.Kobieta)</pre>

	<pre> return string.Format("Pani {3} {0} {1} urodzona w {2}", Imie, Nazwisko, RokUrodzenia, zwrocTytul()); return string.Format("Pan {3} {0} {1} urodzony w {2}", Imie, Nazwisko, RokUrodzenia, zwrocTytul()); } </pre>
5. Do bieżącego projektu dodaj klasę Stypendysta	<ul style="list-style-type: none"> • Z menu wybierz Project -> Add Class. • W oknie dialogowym Add New Item – Osoby z listy Visual Studio installed templates wybierz Class. • W polu Name wpisz Stypendysta. • Kliknij OK. • Klasę Stypendysta uczyni publiczną i zaznacz, że dziedziczy po klasie Student: <pre> public class Stypendysta : Student { } </pre> • Do klasy Stypendysta dodaj żądane pole i właściwość, przy pomocy której możemy uzyskać do niego dostęp. <pre> public decimal Stypendium { set; get; } </pre> • Do klasy Stypendysta dodaj publiczny konstruktor inicjalizujący jej wszystkie pola: <pre> public Stypendysta (string imie, string nazwisko, int rokUrodzenia, Plec plec, int numerIndeksu, decimal stypendium) : base(imie, nazwisko, rokUrodzenia, plec, numerIndeksu) { Stypendium = stypendium; } </pre> • Do klasy Stypendysta dodaj metodę zwracającą w postaci napisu pełną informację na temat obiektu klasy Stypendysta: <pre> public new string ZwrocInformacje() { return string.Format("{0} ma przyznane stypendium w " + "wysokości {1:C}", base.ZwrocInformacje(), Stypendium); } </pre>
6. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. • W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. • W polu Name wpisz TestOsob. • Kliknij OK.
7. Zaznacz projekt TestOsob jako projekt startowy	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestOsob, a następnie z menu kontekstowego wybierz Set as StartUp Project.
8. Do projektu TestOsob dodaj odwołanie do projektu Osoby	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projekt TestOsob, a następnie z menu kontekstowego wybierz Add Reference. • W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt Osoby, a następnie kliknij przycisk OK.
9. Przetestuj klasy Osoba, Student, Stypendysta oraz Wykladowca	<ul style="list-style-type: none"> • Przejdź do pliku Program.cs. • Na górze pliku Program.cs zaimportuj przestrzeń nazw Osoby: <pre> using Osoby; </pre> • Do metody Main dodaj kod, który przetestuje, czy klasa Osoba działa prawidłowo. Przykładowy kod jest zamieszczony poniżej:

	<pre>static void Main(string[] args) { Osoba o1 = new Osoba("Jan", "Kowalski", 1988, Plec.Mezczyzna); Console.WriteLine("Utworzyłeś osobę: {0}", o1.ZwrocInformacje()); Student s1 = new Student("Tomasz", "Nowak", 1989, Plec.Mezczyzna, 1234); Console.WriteLine("Utworzyłeś studenta: {0}", s1.ZwrocInformacje()); Stypendysta st1 = new Stypendysta("Joanna", "Zielińska", 1987, Plec.Kobieta, 1235, 500); Console.WriteLine("Utworzyłeś stypendystę: {0}", st1.ZwrocInformacje()); Wykladowca w1 = new Wykladowca("Maria", "Skłodowska-Curie", 1867, Plec.Kobieta, Wykladowca.Tytuly.prof); Console.WriteLine("Utworzyłeś wykładowcę: {0}", w1.ZwrocInformacje()); Console.ReadKey(); }</pre> <ul style="list-style-type: none"> Na końcu metody Main dodaj następujący kod: <pre>Osoba o2 = s1, o3 = st1, o4 = w1; Console.WriteLine("Utworzyłeś osobę: {0}", o2.ZwrocInformacje()); Console.WriteLine("Utworzyłeś osobę: {0}", o3.ZwrocInformacje()); Console.WriteLine("Utworzyłeś osobę: {0}", o4.ZwrocInformacje());</pre> <p> Z której klasy zostanie wywołana metoda ZwrocInformacje w powyższym kodzie? Dlaczego?</p>
10. Skompiluj i uruchom program	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 90 min)

Twoja firma opracowuje program do zarządzania zasobami ludzkimi dla pewnej firmy. Rodzaje osób, które należy uwzględnić w programie, to: pracownik fizyczny, pracownik umysłowy, kierownik, członek zarządu, praktykant. Poszczególne osoby charakteryzują się następującymi cechami:

- **Pracownik fizyczny** – imię, nazwisko, rok urodzenia, stawka godzinowa, liczba przepracowanych godzin, liczba nadgodzin, informacje na temat kierownika, informacje o umiejętnościach (np. że jest spawaczem i ma uprawnienia na wózek widłowy).
- **Pracownik umysłowy** – imię, nazwisko, rok urodzenia, pensja miesięczna (ryczałt), procent premii, informacje na temat kierownika, numer telefonu, numer pokoju.
- **Kierownik** – imię, nazwisko, rok urodzenia, pensja miesięczna (ryczałt), procent premii, kwota dodatku kierowniczego, numer telefonu, numer telefonu komórkowego, numer pokoju, nazwa działu, którego jest kierownikiem.
- **Członek zarządu** – imię, nazwisko, rok urodzenia, pensja miesięczna (ryczałt), informacje na temat asystenta, ile razy uczestniczył w spotkaniach rady nadzorczej.
- **Praktykant** – imię, nazwisko, rok urodzenia, informacje na temat opiekuna, informacja czy ma przyznane stypendium.

Twoim zadaniem jest opracowanie i zaimplementowanie odpowiedniej hierarchii klas. Oprócz powyższych cech, każda klasa powinna zawierać:

- zestaw odpowiednich konstruktorów
- zestaw właściwości umożliwiające zmianę i odczyt odpowiednich pól
- metody zwracającej w postaci napisu pełną informację o danej osobie
- metody obliczającej pobory miesięczne danej osoby

Pensja dla poszczególnych typów osób jest obliczana w następujący sposób:

- **Pracownik fizyczny** – $\text{stawka godzinowa} \cdot \text{liczba przepracowanych godzin} + 1,5 \cdot \text{stawka godzinowa} \cdot \text{liczba nadgodzin}$.
- **Pracownik umysłowy** – $\text{pensja miesięczna} + \text{procent premii} \cdot \text{pensja miesięczna}$.
- **Kierownik** – $\text{pensja miesięczna} + \text{procent premii} \cdot \text{pensja miesięczna} + \text{kwota dodatku kierowniczego}$.
- **Członek zarządu** – $\text{pensja miesięczna} + \text{liczba spotkań rady nadzorczej} \cdot \text{kwota przyznana za uczestnictwo w spotkaniu rady nadzorczej}$.
- **Praktykant** – w zależności od tego, czy praktykant ma przyznane stypendium, metoda powinna zwracać wartość zero lub kwotę stypendium, która dla każdego praktykanta jest taka sama.

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 8

Wersja 2

Polimorfizm i funkcje wirtualne

Spis treści

Polimorfizm i funkcje wirtualne	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	9
Porady praktyczne	15
Uwagi dla studenta	16
Dodatkowe źródła informacji	16
Laboratorium podstawowe	18
Problem 1 (czas realizacji 20 minut)	18
Problem 2 (czas realizacji 10 minut)	20
Problem 3 (czas realizacji 15 minut)	22
Laboratorium rozszerzone	24
Zadanie 1 (czas realizacji 90 min)	24

Informacje o module

Opis modułu

W tym module zapoznasz się z kolejnym z głównych pojęć programowania obiektowego, a mianowicie polimorfizmem. Nauczysz się, jak definiować metody wirtualne. Poznasz też pojęcie późnego wiązania. Dowiesz się, jakie metody typu object można nadpisać i jak to zrobić. Poznasz kolejne wzorce projektowe, o nazwach metoda fabrykująca i adapter.

Cel modułu

Celem modułu jest pokazanie, co to jest polimorfizm i funkcje wirtualne oraz jak skorzystać z tych mechanizmów w języku C#.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- znał pojęcie polimorfizmu
- potrafił definiować metody wirtualne
- potrafił nadpisać metody wirtualne w klasach pochodnych
- znał metody typu object i potrafił je nadpisać we własnych klasach
- znał różnicę między przesłonięciem a nadpisaniem metody
- rozumiał wzorce projektowe metody fabrykującej i adaptera

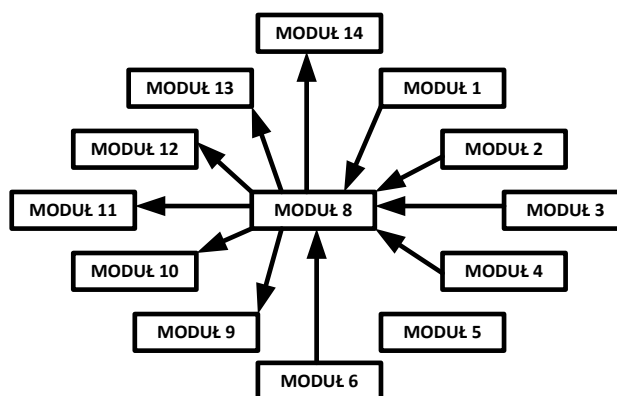
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinienś:

- potrafić definiować klasę
- wiedzieć, co to jest pole i co to jest metoda
- znać modyfikatory dostępu (public i private)
- znać pojęcie konstruktora i potrafić go definiować

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na Rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w modułach „Pojęcie klasy”, „Konstruktor”, „Właściwości i indeksatory” oraz „Składowe statyczne” i „Dziedziczenie”.



Rys. 9 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Postanowiłeś z kolegami napisać grę – wyścigi samochodowe. W grze ścigacie się wybranym modelem samochodu, np. maluchem.

Po pewnym czasie doszliście do wniosku, że grę chcecie uatrakcyjnić dodając nowe modele samochodów: Ferrari i BMW. Stworzyliście klasy reprezentujące nowe modele samochodów. Wszystkie klasy reprezentujące poszczególne typy pojazdów mają podobny zbiór składowych, w końcu każda z nich reprezentuje samochód. Teraz przystępujecie do modyfikacji samej gry, aby uwzględniła nowe pojazdy. Niestety okazuje się, że w zasadzie musicie napisać ją od nowa.

Pewnie zadajesz sobie pytanie, czy istnieje sposób takiego napisania gry, aby dodanie nowego modelu samochodu wiązało się tylko z niewielkimi modyfikacjami silnika programu. Oczywiście, że tak. Wystarczy, że silnik gry napiszesz używając klasy bazowej Samochod. Każda klasa reprezentująca nowy model pojazdu będzie dziedziczyła po tej klasie. W samej grze wystarczy tylko zmienić miejsce, gdzie wybierany i tworzony jest dany model samochodu. Zmienne klasy bazowej mogą zawierać referencję do obiektów klas pochodnych. W pozostałej części programu posługujesz się tylko zmienną klasy Samochod.

Prawdopodobnie dopadły Cię następujące wątpliwości: „Skoro posługuję się zmienną klasy Samochod, to cały czas jeżdżę tylko tym podstawowym samochodem. Wywoływane są metody z klasy Samochod, ponieważ posługuję się zmienną klasy bazowej. Gdzie zmiana zachowania i wyglądu samochodu dostarczona w klasach pochodnych?” Jeżeli masz powyższe wątpliwości, zapewne nie słyszałeś o funkcjach wirtualnych. Stoi zatem przed tobą niepowtarzalna okazja zawarcia bliższej znajomości z metodami wirtualnymi oraz poznania, jak projektować i tworzyć łatwo rozszerzalne aplikacje. Pytasz jak? Przeczytaj ten moduł.

Podstawy teoretyczne

Polimorfizm jest to wykazywanie różnej funkcjonalności podczas wywoływania metody, w zależności od typu obiektu, na rzecz którego wywołana została dana metoda. Nie ważny jest typ zmiennej, za pomocą której wywołujemy daną metodą. W języku C# do implementacji polimorfizmu używamy metod wirtualnych oraz mechanizmu dziedziczenia. Do utworzenia metody wirtualnej używamy słowa `virtual`:

```
class Bazowa {  
    public virtual void Metoda() {  
    }  
}
```

W celu utworzenia nowej wersji metody w klasie pochodnej, czyli *nadpisanie metody* (ang. *method override*), stosujemy słowo kluczowe `override`:

```
class Pochodna1 : Bazowa {  
    public override void Metoda() {  
    }  
}  
  
class Pochodna2 : Bazowa {  
    public override void Metoda() {  
    }  
}
```

Mając tak zdefiniowaną hierarchię klas oraz następujący kod:

```
Bazowa b = new Bazowa();
char c = Console.ReadKey().KeyChar;
switch(c) {
    case 'a': b = new Pochodna1();
              break;
    case 'b': b = new Pochodna2();
              break;
}
b.Metoda();
```

nie jesteśmy w stanie określić, która metoda zostanie wywołana. Zależy to od klawisza wciśniętego przez użytkownika. Jeżeli użytkownik wybierze klawisz **a**, zostanie wywołana metoda zdefiniowana w klasie *Pochodna1*. W przypadku , gdy użytkownik naciśnie klawisz **b**, wybrana będzie metoda zdefiniowana w klasie *Pochodna2*. W pozostałych przypadkach będzie wywołana metoda z klasy *Bazowa*. Można powiedzieć, że decyzja, która metod ma być wywołana, jest podejmowana w czasie działania programu, a nie w czasie kompilacji.

Zachodzi pytanie jaka magia za tym stoi, czyli jak to jest zaimplementowane. W przypadku metod wirtualnych wywołanie jest dwuetapowe i polega na wykorzystaniu pewnych dodatkowych informacji. Każdy obiekt posiada pole zawierające referencje do tzw. *tablicy funkcji wirtualnych* (ang. *vtable*). Każda klasa posiada własny pojedynczy egzemplarz tej tablicy, który jest współdzielony przez wszystkie obiekty danej klasy. Elementami tablicy funkcji wirtualnych są adresy w pamięci, gdzie znajdują się definicje poszczególnych metod. Element reprezentujący daną metodę wirtualną ma stały indeks w poszczególnych tablicach funkcji wirtualnych. Mając zdefiniowane następujące klasy:

```
class Bazowa {
    public virtual void F() {
    }
}

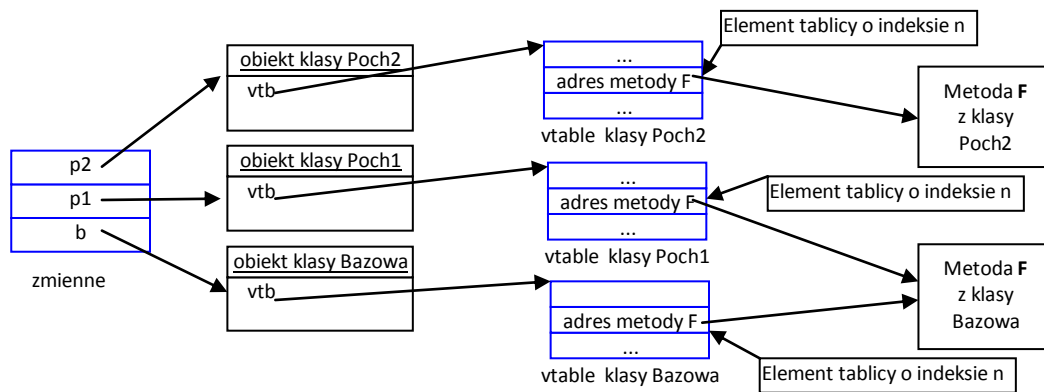
class Poch1 : Bazowa {
}

class Poch2 : Bazowa {
    public override void F() {
    }
}
```

oraz odpowiednio utworzone obiekty tych klas:

```
Bazowa b = new Bazowa();
Bazowa p1 = new Poch1();
Bazowa p2 = new Poch2();
```

implementację polimorfizmu możemy spróbować przybliżyć przy pomocy rys. 10 (pole zawierające referencje do tablicy funkcji wirtualnych nazwijmy *vtb*).



Rys. 10 Dwuetapowe wywołanie metody wirtualnej

W celu wywołania metody wirtualnej, zamiast bezpośredniego skoku do adresu w pamięci, gdzie zdefiniowana jest dana metoda, musimy najpierw określić ten adres. Adres ten jest odczytywany z tablicy funkcji wirtualnych skojarzoną z klasą, która definiuje typ obiektu, na rzecz którego dana metoda jest wywoływana. Powtórzmy, nie brany jest typ zmiennej, przy pomocy której wywołujemy daną metodę, tylko typ obiektu. Dopiero po określeniu adresu następuje skok do miejsca w pamięci, gdzie zdefiniowana jest żądana funkcja.

Przesłonięcie kontra nadpisanie

Z modułu 6 tego kursu wiemy, że metody klasy bazowej możemy przesłonić przy pomocy słowa `new`. Zachodzi pytanie, czy metody wirtualne też mogą być przesłonięte i która metoda jest wtedy wywoływana. Odpowiedź na pierwsze pytanie brzmi tak, a jak to działa, rozważmy na następującym przykładzie. Załóżmy, że mamy zdefiniowaną następującą hierarchię klas:

```
class A {
    public virtual void F() {
    }
}

class B : A {
    public override void F() {
    }
}

class C : B {
    public new virtual void F() {
    }
}

class D : C {
    public override void F() {
    }
}
```

Możemy zauważyć, że w powyższy, przykładzie klasa B dziedziczy po klasie A i nadpisuje metodę F klasy A. Klasa C dziedziczy po klasie B i przesłania metodę F klasy bazowej. Słowo `virtual` nie jest konieczne przy przesłanianiu metod wirtualnych i jest użyte na potrzeby tego przykładu. Klasa D dziedziczy po klasie C i nadpisuje metodę F z klasy bazowej. Rozważmy następujący kod:

```
A a1 = new A();
a1.F();
```

W powyższy kodzie, typ zmiennej i typ obiektu jest identyczny i jest określony przez klasę A, więc na pewno zostanie wywołana metoda z klasy A. Poniższy przykład powinien być również oczywisty:

```
A a2 = new B();
a2.F();
```

Metoda F jest metodą wirtualną i jest nadpisana w klasie B, a w przypadku metod wirtualnych ważny jest typ obiektu, a nie typ zmiennej, więc zostanie wywołana metoda F zdefiniowana w klasie B.

Idźmy dalej i zastanówmy się, co będzie wywołane w następującym przypadku:

```
A a3 = new C();
a3.F();
```

W powyższym przykładzie zostanie wywołana metoda klasy B. Może się to wydawać dziwne, gdyż ani zmienna, ani obiekt nie jest typu definiowanego przez klasę B. Spowodowane jest to tym, że użycie słowa `new` powoduje niejako utworzenie zupełnie nowej funkcji, nie związanej z metodą z klasy bazowej. Można przyjąć, że nadpisanie tworzy nową wersję metody klasy bazowej, natomiast przesłonięcie tworzy nową metodę i przerywa mechanizm wirtualności.

Dodanie słówka `virtual` obok słowa `new` powoduje, że w tablicy funkcji wirtualnych definiowanej klasy zostanie utworzony nowy element reprezentujący nową metodę.

Następny przykład nie powinien chyba już stanowić problemu w określeniu, z której klasy metoda ma być wywołana:

```
A a4 = new D();
a4.F();
```

Mechanizm wirtualności dla metody z klasy A, na „ścieżce dziedziczenia” od klasy A do klasy D został przerwany w klasie C. Zostanie więc wywołana metoda z klasy bazowej klasy C, czyli klasy B, jako „najnowsza” wersja metody z klasy A.

Określenie metody, która będzie wywołana w następującym kodzie powinno być oczywiste:

```
C c1 = new D();
c1.F();
```

Zostanie wywołana metoda zdefiniowana w klasie D. W powyższym przykładzie chodzi nam o wywołanie najnowszej wersji metody z klasy C na „ścieżce dziedziczenia” od klasy C do klasy D.

Nadpisanie metod typu object

Każda typ dziedziczy po typie `object`, dlatego każdy z nich zawiera metody zdefiniowane w typie `object`. Do metod tych należą:

- `GetType` – metoda zwracająca informacje o typie danego obiektu lub zmiennej, szczegóły na temat tej metody zostaną omówione w module 13 tego kursu.
- `MemberwiseClone` – metoda służąca do wykonania płytkiej kopii obiektu. Jest to składowa chroniona.
- `ReferenceEquals` – metoda służąca do sprawdzenia, czy dwie zmienne zawierają referencję do tego samego obiektu. Jest to składowa statyczna.
- `Finalize` – metoda wirtualna wywoływana przez Garbage Collector przed zwolnieniem pamięci zajmowanej przez dany obiekt. Szczegóły na temat tej metody zostaną omówione w module 11 tego kursu. Jest to składowa chroniona.

oraz trzy publiczne wirtualne metody które są głównym tematem tego rozdziału:

- `ToString`
- `Equals` (istnieje też wersja statyczna tej metody, przyjmująca jako argumenty dwie zmienne typu `object`)

- GetHashCode

ToString

Domyślna wersja metody ToString zwraca w pełni kwalifikowaną nazwę klasy. W przypadku gdy chcemy, aby metoda przekazywała stan danego obiektu, musimy nadpisać tę metodę dla danej klasy.

Metoda ta jest wywoływana np. przy budowaniu napisu, stosując tzw. *złożone formatowanie* (ang. *Composite Formatting*). Złożone formatowanie jest wykorzystywane między innymi przez metody:

- String.Format
- Console.WriteLine
- StringBuilder.AppendFormat

Mając zdefiniowaną klasę

```
namespace Test1 {  
    class K1 {  
    }  
}
```

następujący kod:

```
K1 k = new K1();  
Console.WriteLine(k);
```

spowoduje wypisanie na ekranie:

```
Test1.K1
```

Nadpisanie metody ToString:

```
namespace Test1 {  
    class K1 {  
        public override string ToString() {  
            return "Metoda nadpisana";  
        }  
    }  
}
```

spowoduje, że w wyniku działania poprzedniego kodu otrzymamy na ekranie napis:

```
Metoda nadpisana
```

Należy zwrócić uwagę, że metoda ToString będzie wywołana w przypadku złożonego formatowania pod warunkiem, że dana klasa nie implementuje interfejsów ICustomFormatter lub IFormattable. Interfejsy zostaną omówione w module 9 tego kursu. Na temat złożonego formatowania można znaleźć więcej w MSDN pod tematem „Composite Formatting”.

Equals

Metoda Equals służy do określenia, czy dwa obiekty są sobie równe. Nienadpisana wersja tej metody dla typów referencyjnych porównuje same referencje, czyli zwraca wartość true tylko wtedy, gdy zmienna, przy pomocy której wywołujemy tą metodę i zmienna przekazana do tej metody zawierają odwołanie do tego samego obiektu. Dla typów bezpośrednich jest sprawdzane, czy obie zmienne mają tę samą reprezentację bitową. Porównanie obiektów według wartości jego pól wymaga nadpisania tej metody. Samo pojęcie równości leży w gestii programisty nadpisującego metodę Equals w danej klasie lub strukturze.

Nadpisując metodę Equals należy wyróżnić następujące kroki:

- sprawdzenie, czy zmienna przekazywana jest różna od null

- sprawdzanie, czy typ zmiennej przekazywanej jest taki sam, jak typ zmiennej na rzecz której wywoływana jest metoda `Equals`
- rzutowanie zmiennej przekazywanej do metod, na typ określony przez klasę, którą implementujemy, w celu uzyskania dostępu do pól obiektu implementowanej klasy
- porównanie wartości odpowiednich pól

Przykładowa implementacja metody `Equals` uwzględniając wszystkie powyższe kroki jest przedstawiona poniżej:

```
class K1 {
    private TypReferencyjny pole1;
    private TypBezposredni pole2
    ...
    public override bool Equals(Object obj) {
        if( obj == null) return false;
        if( this.GetType() != obj.GetType() ) return false;
        K1 k = (K1) obj;    //rzutowanie
        if( !Object.Equals(pole1, k.pole1) ) return false;
        if( !pole2.Equals(k.pole2) ) return false;
        return true;
    }
}
```

Dla pól typu referencyjnego została użyta metoda statyczne `Equals` typu `object`, ponieważ pole może mieć wartość `null`. Metoda ta w przypadku, gdy oba argumenty mają wartość `null`, zwraca wartość `true`.

Metoda `GetType` i jak pobrać informację o typie obiektu w czasie działania programu jest dokładniej opisana w module 13 tego kursu. O operatorach rzutowania z klasy pochodnej do klasy podstawowej można dowiedzieć się więcej w module 9.

W przypadku, gdy dana klasa dziedziczy po klasie, w której jest nadpisana metod `Equals`, należy wywołać metodę klasy bazowej przy pomocy słowa kluczowego `base` w celu sprawdzenia równości części odziedziczonej.

GetHashCode

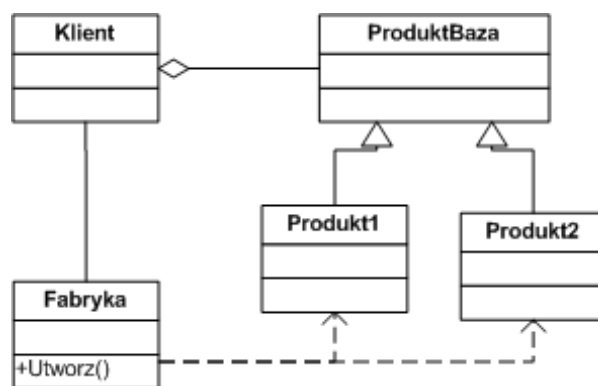
Nadpisując metodę `Equals` powinniśmy również nadpisać metodę `GetHashCode`. Metoda ta zwraca wartość typu `int` reprezentującą *wartość skrótu* (ang. *hash code*) dla danego obiektu. Nadpisując metodę `GetHashCode` powinniśmy przestrzegać następujących zasad:

- Przy generacji skrótu metoda powinna uwzględnić wartość co najmniej jednego pola niestatycznego danej klasy.
- W przypadku, gdy dwa obiekty są równe (metoda `Equals` zwraca `true`), metoda `GetHashCode` powinna zwracać ten sam skrót.
- Metoda nie powinna zgłaszać wyjątków.

Metoda `GetHashCode` jest wykorzystywana w klasie `Hashtable` i klasach po niej dziedziczących.

Wzorzec projektowy metody fabrykującej

Jak już było wspomniane we wstępie, idealnie byłoby uzyskanie kodu otwartego na rozbudowę, a zamkniętego na modyfikację. Funkcje wirtualne dają nam taką możliwość, należy tylko odpowiednio zaprojektować swoją aplikację. Sposób jak projektować takie aplikacje pokazuje wzorzec *metoda fabrykująca* (ang. *factory metod*). Podstawą tego wzorca jest metoda, która na podstawie przekazanych informacji konstruuje obiekt odpowiedniego typu i zwraca go. Typ konstruowanego obiektu jest wybierany na podstawie podanej informacji spośród dostępnych klas pochodnych wybranej klasy. Diagram obrazujący wzorzec metody fabrykującej jest przedstawiony na rysunku rys. 11.

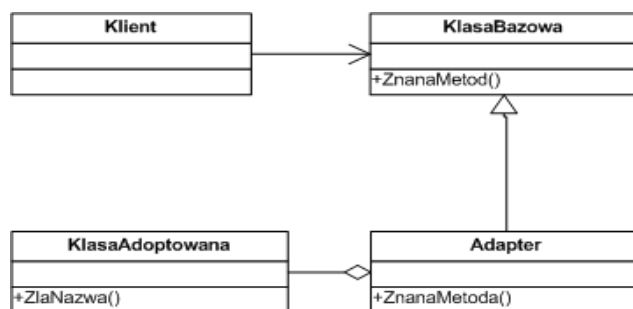


Rys. 11 Wzorzec projektowy metoda fabrykująca

Na diagramie (rys. 11) klient zawiera pole typu określonego przez klasę ProduktBaza (kreska zakończona rombem oznacza agregację – obiekt klasy klient składa się między innymi z obiektu klasy ProduktBaza). Klasy Produkt1 i Produkt2 są klasami pochodnymi klasy ProduktBaza. Klient sam nie tworzy obiektu klasy ProduktBaza lub klas po niej dziedziczących. Za utworzenie obiektu odpowiedniego typu jest odpowiedzialna klasa Fabryka i jej metoda Utworz. Klient nie musi być do końca świadomy, z jakim dokładnie typem obiektu współpracuje. Przykład kodu stosującego wzorzec metoda fabrykująca jest przedstawiony w rozdziale „Przykładowe rozwiązanie” w dalszej części tego modułu.

Wzorzec projektowy adapter

Wzorzec projektowy adapter dopasowuje istniejącą klasę (np. utworzoną przez innego programistę) do istniejącej hierarchii klas. Załóżmy, że mamy hierarchię: klasę KlasaBazowa jako klasę podstawową i klasy Produkt1 i Produkt2 jako klasy pochodne. Mamy też klasę KlasaAdoptowana, która nie dziedziczy po klasie KlasaBazowa. Chcemy korzystać z klasy KlasaAdoptowana, podobnie jak z klas Produkt1 i Produkt2, to jest odwoływać się do obiektów tej klasy przy pomocy zmiennej typu KlasaBazowa. Adapter rozwiązuje ten problem przez utworzenie klasy opakowującej klasę KlasaAdoptowana, czyli posiadającej pole typu KlasaAdoptowana oraz dziedziczącej po klasie KlasaBazowa. Nadpisane metody klasy KlasaBazowa w klasie opakowującej wywołują odpowiednie metody z klasy KlasaAdoptowana w celu uzyskania żądanej funkcjonalności. Diagram obrazujący wzorzec adapter jest przedstawiony na rys. 12.



Rys. 12 Wzorzec projektowy adapter

Przykład kodu stosującego wzorzec adapter jest przedstawiony w rozdziale „Przykładowe rozwiązanie” w dalszej części tego modułu.

Przykładowe rozwiązanie

Nadpisywanie metod typu object

Założmy, że mamy zdefiniowaną klasę reprezentującą informację o książce w następujący sposób:

```
class Autor {
    private string imie;
    private string nazwisko;

    public Autor(string imie, string nazwisko) {
        if (string.IsNullOrEmpty(imie) || string.IsNullOrEmpty(nazwisko))
            throw new ArgumentException("Imie i nazwisko nie może być puste");
        this.imie = imie;
        this.nazwisko = nazwisko;
    }

    public string Imie { get { return imie; } }
    public string Nazwisko { get { return nazwisko; } }
}

class Ksiazka {
    public Autor Autor { set; get; }
    public string Tytul { set; get; }
    public int RokWydania { set; get; }
    public string Wydawnictwo { set; get; }
    public string Isbn { set; get; }
}
```

Naszym zadaniem jest nadpisanie metod typu object w klasie Ksiazka.

Nadpisanie metody ToString

Nadpisana metoda ToString powinna zwracać aktualny stan obiektu klasy Ksiazka w formie napisu:

```
public override string ToString() {
    return string.Format("{0} {1}., \"{2}\", {3}, {4}, {5}",
        Autor.Nazwisko, Autor.Imie[0], Tytul, Wydawnictwo,
        RokWydania, Isbn);
}
```

Nadpisanie metody Equals

Nadpisując metodę Equals zakładamy, że jeżeli dwa obiekty klasy Ksiazka mają ten sam numer ISBN, to reprezentują tę samą książkę. Nie wszystkie książki posiadają jednak numer ISBN. Dla książek, które nie posiadają tego unikalnego numeru, aby dwa obiekty reprezentowały tę samą książkę, wszystkie pozostałe pola muszą być równe. W przypadku porównywania pól, które są napisami, możemy użyć operatorów == lub !=, ponieważ typ string ma je przeciążone. Operatory te korzystają z metody Equals klasy string. Nie tylko sprawdza ona, czy zmienne wskazują na ten sam obiekt, ale również sprawdza identyczność napisów. Dla pola określonego przez klasę Autor, w celu porównania dwóch pól użyjemy metody statycznej Equals typu object. Zwróci ona wartość true w przypadku, gdy oba pola wskazują na ten sam obiekt lub oba mają wartość null:

```
public override bool Equals(object obj) {
    if( obj == null) return false;
    if( this.GetType() != obj.GetType() ) return false;
    Ksiazka k = (Ksiazka) obj;
    if (Isbn != null || k.Isbn != null) {
        if (Isbn == k.Isbn)
            return true;
        else
            return false;
    }
    if( !string.Equals(Tytul, k.Tytul) ) return false;
    if( !Autor.Equals(Autor, k.Autor) ) return false;
}
```

```
        if (RokWydania != k.RokWydania) return false;
        if (Wydawnictwo != k.Wydawnictwo) return false;
        return true;
    }
```

Nadpisanie metody GetHashCode

Przypomnijmy, że nadpisując metodę Equals powinniśmy również nadpisać metodę GetHashCode, aby zachować implikację, że jeżeli dwa obiekty są równe (metoda Equals zwraca true) to metoda GetHashCode powinna zwrócić tę samą wartość skrótu:

```
    public override int GetHashCode() {
        if (Isbn != null)
            return Isbn.GetHashCode();
        return ToString().GetHashCode();
    }
```

Wzorzec metody fabrykującej

Naszym zadaniem jest utworzenie programu edukacyjnego, który wyświetla informacje o różnych zwierzętach. Wśród informacji, które dostarcza program, są dźwięki jakie wydaje dane zwierzę oraz jego nazwa łacińska. Kolejne zwierzęta będą dostarczane wraz z rozwojem oprogramowania. W projekcie aplikacji wykorzystamy wzorzec metody fabrykującej.

Utworzenie podstawowej hierarchii klas

Utworzymy klasę bazową, która będzie zawierała funkcjonalność wymaganą przez nasz program. Funkcjonalność będzie zaimplementowana przy pomocy metod wirtualnych, które będą nadpisywane w klasach pochodnych reprezentujących poszczególne zwierzęta:

```
class Zwierze {
    public virtual void WydadzGlos() {
        Console.WriteLine("Zwierze wydają różne dźwięki...");
    }

    public virtual string NazwaLacinska {
        get { return "Animalia"; }
    }

    public override string ToString() {
        return "zwierze";
    }
}
```

Mając zdefiniowaną klasę bazową możemy utworzyć klasę reprezentującą wybrane zwierzę:

```
class Owca : Zwierze {
    public override void WydadzGlos() {
        Console.WriteLine("Beee...");
    }

    public override string NazwaLacinska {
        get { return "Ovis aries"; }
    }

    public override string ToString() {
        return "owca";
    }
}
```

Utworzenie klasy fabryki

Kolejnym naszym krokiem będzie utworzenie metody tworzącej obiekt reprezentujący żądane zwierzę. Przykładowa implementacja znajduje się poniżej:

```
static class Fabryka {
    public static Zwierze Utworz(int i) {
        Zwierze zwierze = null;
        switch (i) {
            case 1: zwierze = new Zwierze();
                    break;
            case 2: zwierze = new Owca();
                    break;
        }
        return zwierze;
    }
}
```

Metoda Utworz tworzy zwierzę określonego typu, w zależności od wartości parametru przekazanego do metody. Jest to kluczowa metoda wzorca metody fabrykującej. Dodatkowo do klasy Fabryka dodajmy metodę, która wyświetli dostępne opcje użytkownikowi:

```
public static int Menu() {
    Console.Clear();
    Console.WriteLine("\n\t\t\t1 - Informacje ogólne o zwierzętach");
    Console.WriteLine("\n\t\t\t2 - Informacje o owcy");
    Console.WriteLine("\n\t\t\t0 - Koniec");
    int i ;
    bool b;
    do {
        do {
            b = int.TryParse(Console.ReadLine(), out i);
        }
        while (!b);
    }
    while (0 > i || i > 2);
    return i;
}
```

Utworzenie silnika (części głównej) naszego programu

Silnik naszego programu będzie pracował na zmiennej klasy bazowej Zwierze. W celu utworzenia obiektu konkretnej klasy będzie używał metody Utworz klasy Fabryka. Przykładowa implementacja znajduje się poniżej:

```
class Klient {
    private Zwierze zwierze;

    public int Menu() {
        Console.WriteLine("Twój aktualny wybór to {0}\n", zwierze);
        Console.WriteLine("1 - Dźwięki wydawane przez zwierzę");
        Console.WriteLine("2 - Nazwa łacińska");
        Console.WriteLine("3 - Powrót do menu główne");
        int i;
        bool b;
        do {
            b = int.TryParse(Console.ReadLine(), out i);
        }
        while (!b);
        return i;
    }
}
```



```
public void Uruchom() {
    int i, j;
    while(true) {
        i = Fabryka.Menu();
        if(i == 0)
            break;
        zwierze = Fabryka.Utworz(i);
        do {
            j = Menu();
            Console.Clear();
            switch (j) {
                case 1: zwierze.WydajGlos();
                        Console.ReadKey();
                        break;
                case 2: Console.WriteLine(zwierze.NazweLacinska);
                        Console.ReadKey();
                        break;
            }
        } while (j != 3);
    }
}
```

Zwróćmy uwagę, że klasa klient posługuje się tylko klasą bazową. Nigdzie w silniku naszego programu nie wykorzystujemy nazwy klasy pochodnej. Można powiedzieć, że dla klasy Klient nie jest istotny określenie dokładnego typu pola zwierze.

Na koniec dodajmy kod do metody Main:

```
static void Main(string[] args) {
    Klient k = new Klient();
    k.Uruchom();
}
```

Dodanie nowej klasy do programu

Załóżmy, że po pewnym czasie została utworzona nowa klasa reprezentująca kolejne zwierzę:

```
class Osioł : Zwierze {
    public override void WydajGlos() {
        Console.WriteLine("Ioo ioo...");
    }

    public override string NazweLacinska {
        get { return "Equus asinus"; }
    }

    public override string ToString() {
        return "osioł";
    }
}
```

W celu użycia nowej klasy w naszym programie, wystarczy zmienić tylko kod metod klasy Fabryka. W tym celu odpowiednio modyfikujemy metodę Utworz:

```
public static Zwierze Utworz(int i) {
    Zwierze zwierze = null;
    switch (i) {
        ...
        case 3: zwierze = new Osioł();
                break;
    }
```

```
    }  
    return zwierze;  
}
```

oraz metodę Menu:

```
public static int Menu() {  
    ...  
    Console.WriteLine("\n\t\t\t3 - Informacje o osłach");  
    ...  
    do {  
        ...  
    }  
    while (0 > i || i > 3);  
    return i;  
}
```

Zwróćmy uwagę, że nie musimy modyfikować żadnej linijki w silniku naszego programu. Otrzymaliśmy kod zamknięty. Jednocześnie możemy dodawać nową funkcjonalność. Ilość koniecznych modyfikacji jest niewielka, a co za tym idzie przetestowanie nowego programu powinno być w miarę proste.

Wzorzec adapter

Załóżmy, że do programu z przykładu 2 chcemy dodać klasę reprezentującą nowe zwierzę. Niestety nowa klasa nie dziedziczy po naszej klasie bazowej, a nie możemy modyfikować jej kodu, bo jest to zbyt kosztowne. W tym przykładzie zastosujemy wzorzec projektowy adapter, aby dostosować klasę do naszych potrzeb.

Adaptacji klasy

Założmy, że nowo otrzymana klasa wygląda następująco:

```
public class Wilk {  
    public string Wyj() {  
        return "Auuu...";  
    }  
  
    public string PodajNazweLacinska() {  
        return "Canis lupus";  
    }  
}
```

Klasę tę możemy zaadoptować w następujący sposób:

```
class WilkZaadoptowany : Zwierze {  
    private Wilk wilk = new Wilk();  
  
    public override void WydajGlos() {  
        Console.WriteLine(wilk.Wyj());  
    }  
  
    public override string NazweLacinska {  
        get { return wilk.PodajNazweLacinska(); }  
    }  
  
    public override string ToString() {  
        return "wilk";  
    }  
}
```

Zastosowanie nowej klasy w programie

Sposób użycia nowo utworzonej przez nas klasy jest analogiczny do tego, jakiego użyliśmy we wcześniejszym przykładzie. Musimy tylko zmodyfikować odpowiednio metody klasy Fabryka:

```
public static int Menu() {  
    ...  
    Console.WriteLine("\n\t\t\t4 - Informacje o wilkach");  
    ...  
    do {  
        ...  
    }  
    while (0 > i || i > 4);  
    return i;  
}  
  
public static Zwierze Utworz(int i) {  
    Zwierze zwierze = null;  
    switch (i) {  
        ...  
        case 4: zwierze = new WilkZaadoptowany();  
            break;  
    }  
    return zwierze;  
}
```

Gotowe rozwiązanie powyższych przykładów znajduje się w katalogu **Demo\Modul08**.

Porady praktyczne

- W przypadku struktur nie możesz definiować metod wirtualnych, ponieważ nie można po nich dziedziczyć. W typach strukturalnych przez siebie definiowanych możesz natomiast nadpisywać metody wirtualne typu `object`.
- W klasach pochodnych nie musisz nadpisywać wszystkich metod wirtualnych klasy podstawowej.
- Metoda z przydomkiem `override` jest też metodą wirtualną. Nie możesz jednak stosować razem słowa `virtual` i `override`.
- Pominięcie słowa `override` przy definicji metody w klasie pochodnej powoduje, że metoda zostanie przesłonięta, a nie nadpisana. Jeżeli metoda w klasie bazowej i klasie pochodnej ma taką samą sygnaturę i nie zaznaczymy w sposób jawny, czy chcemy metodę nadpisać czy przesłonić, kompilator doda domyślnie do definicji metody w klasie pochodnej słowo `new`, czyli przesłoni metodę.
- Typ `string` ma nadpisaną metodę `Equals`, która sprawdza, czy zmienne zawierają referencje do obiektów, które reprezentują te same napisy. Metoda ta jest używana przez operatory `==` oraz `!=`, które są przeciążone dla typu `string`.
- Rozważ nadpisanie metody `Equals` w przypadku, gdy dwie różne instancje danego typu mogą być rozważane jako równe.
- Przeciążając operatory `==` i `!=` nadpisuj zawsze metodę `Equals`.
- Nadpisując metodę `Equals` zastanów się, czy nie dostarczyć równolegle wersji, która będzie przyjmować jako argument zmienną, której typ jest definiowany przez daną klasę (klasę, dla której przeciążasz metodę `Equals`). Wersja „typizowana” będzie zapewne wydajniejsza, gdyż nie ma konieczności dokonywania w niej rzutowania z typu `object` na daną klasę.
- W przypadku, gdy nadpisujesz metodę `Equals`, nadpisz również metodę `GetHashCode`.
- Nie używaj wartości kodu skrótu zwróconego przez metodę `GetHashCode` do stwierdzenia równości dwóch obiektów. Obiekty równe powinny mieć tę samą wartość kodu skrótu, ale jest to implikacja, a nie równoważność.

- Staraj się nadpisywać w swoich klasach i strukturach metodę `ToString` w celu dostarczenia tekstowej reprezentacji stanu obiektu danej klasy.
- W przypadku, gdy w nadpisanej metodzie klasy pochodnej chcesz wywołać nadpisaną metodę wirtualną z klasy bazowej, podobnie jak w przypadku przestaniania metody, użyj słowa kluczowego `base`.
- Słowo `virtual` może również występować przy definicji właściwości i zdarzenia.
- Metoda wirtualna nie może być metodą prywatną.
- W języku C# metoda wirtualna nie może być metodą statyczną.
- W przypadku, gdy dla metody wirtualnej w klasie bazowej nie możemy zdefiniować domyślnej implementacji i metoda ta w klasach pochodnych jest opcjonalna, w metodzie klasy bazowej zgłoś wyjątek `NotSupportedException`.
- Definiuj metodę jako wirtualną tylko wtedy, gdy jesteś pewny, że w klasach pochodnych będzie ona nadpisana. Pamiętaj, że wywołanie metody wirtualnej jest wolniejsze niż zwykłej metody.
- Definiując metodę lub właściwość zastanów się, czy nie powinna być ona wirtualna, szczególnie wtedy, gdy planujesz dziedziczyć po klasie, w której jest ona zdefiniowana.
- Stosując wzorzec projektowy metody fabrykującej otrzymasz program, który będzie można w miarę prosty sposób rozszerzyć.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- Wiesz, co to jest polimorfizm
- Wiesz, co to są metody wirtualne, znasz zasadę ich działania i czym różnią się od zwykłych metod
- Wiesz, co to jest tablica funkcji wirtualnych (vtb)
- potrafisz zdefiniować metodę wirtualną i nadpisać ją w klasie pochodnej
- znasz słowa kluczowe języka C# `virtual` i `override`
- rozumiesz różnicę między przestonięciem a nadpisaniem metody
- znasz metody wirtualne typu `object` i wiesz jak je nadpisać
- rozumiesz i potrafisz zastosować wzorzec projektowy metody fabrykującej
- rozumiesz i potrafisz zastosować wzorzec projektowy adapter

Dodatkowe źródła informacji

1. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka programistów chcących nauczyć się programować w języku C#.

2. Stephen C. Perry, *C# i .NET*, Helion, 2006

Książka w porównaniu z poprzednią pozycją skierowana jest do osób trochę bardziej zaawansowanych. Opisuje C# i .NET Framework w wersji 2.0

3. Andrew Troelsen, *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition*, Apress, 2007

Książka przeznaczona jest dla bardziej zaawansowanych programistów. Czwarte wydanie tej książki opisuje język C# 3.0 i platformę .NET 3.5.

4. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad, jak powinniśmy pisać swoje programy.

5. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Wzorce projektowe Wydanie II*, WNT, 2008

Książka, która wprowadziła pojęcie wzorców projektowych do informatyki. Lektura obowiązkowa dla każdego, kto chce poznać temat wzorców projektowych.

6. Steven John Metsker, *C#. Wzorce projektowe*, Helion, 2005

Książka jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia jak wykorzystać cechy języka C# do tworzenia poprawnego kodu poprzez zastosowanie wzorców.

7. Judith Bishop, *C# 3.0 Design Patterns*, O'Reilly Media, Inc. 2008

Książka, podobnie jak poprzednia, jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia również nowe cechy języka C#.

8. Codeguru, <http://www.codeguru.pl>


Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie. Możesz tu również znaleźć artykuły poświęcone wzorcom projektowym i ich implementacji w języku C#.

Laboratorium podstawowe

Problem 1 (czas realizacji 20 minut)

Twoja firma opracowuje program edukacyjny dla pewnej szkoły muzycznej. Wyświetla on informacje o różnych instrumentach. Wśród informacji, które dostarcza program, są przykłady dźwięków, jakie wydaj dany instrument oraz jego krótka charakterystyka. Kolejne instrumenty będą dostarczane wraz z rozwojem oprogramowania. Wersja pierwotna zawiera ogólną informację o instrumentach oraz informację o trąbce. W projekcie aplikacji wykorzystaj wzorzec metoda fabrykująca.

Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition	<ul style="list-style-type: none"> Otwórz Visual C# 2008 Express Edition. Z menu wybierz File -> New Project. Z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz Instrumenty. Kliknij OK. Z menu wybierz File -> Save Instrumenty. W polu Location wybierz folder w którym będzie zapisany projekt. Zaznacz pole wyboru Create directory for solution. W polu Solution Name wpisz Modul08. Naciśnij przycisk Save
2. Do projektu Instrumenty dodaj nową klasę Instrument	<ul style="list-style-type: none"> Z menu wybierz Project -> Add New Item. W oknie dialogowym Add New Item – Instrumenty na liście Template wybierz szablon Class. Następnie w polu Name wpisz Instrument. Kliknij Add.
3. Do klasy Instrument dodaj żądaną funkcjonalność	<ul style="list-style-type: none"> Przejdź do pliku Instrument.cs. Do klasy Instrument dodaj wirtualną metodę Graj: <pre>public virtual void Graj() { Console.WriteLine("Instrumenty wydają dźwięki o różnej " + "częstotliwości i barwie..."); }</pre> Do klasy Instrument dodaj wirtualną właściwość tylko do odczytu Opis: <pre>public virtual string Opis { get { return "Instrument muzyczny - przyrząd wytwarzający dźwięk"; } }</pre> W klasie Instrument nadpisz wirtualną metodę ToString: <pre>public override string ToString() { return "instrument"; }</pre>
4. Do projektu Instrumenty dodaj nową klasę Trabka	<ul style="list-style-type: none"> Z menu wybierz Project -> Add Class. W oknie dialogowym Add New Item – Instrumenty na liście Template wybierz szablon Class. Następnie w polu Name wpisz Trabka. Kliknij Add.
5. Do klasy Trabka dodaj żądaną funkcjonalność	<ul style="list-style-type: none"> Przejdź do pliku Trabka.cs. Zaznacz, że klasa Trabka dziedziczy po klasie Instrument:

	<pre>class Trabka : Instrument</pre> <ul style="list-style-type: none"> W klasie Trabka nadpisz wirtualną metodę Graj: <pre>public override void Graj() { Console.WriteLine("Tra ta ta ta"); }</pre>  Wpisanie na początku słowa override spowoduje, że środowisko podpowie Ci, jakie metody możesz nadpisać. Wybranie nazwy metody z menu kontekstowego spowoduje wygenerowanie jej szkieletu. W klasie Trabka nadpisz wirtualną właściwość Opis: <pre>public override string Opis { get { return "Trąbka jest to instrument dęty blaszany..."; } }</pre> W klasie Trabka nadpisz wirtualną metodę ToString: <pre>public override string ToString() { return "trąbka"; }</pre>
6. Do projektu Instrumenty dodaj nową klasę Fabryka	<ul style="list-style-type: none"> Z menu wybierz Project -> Add Class. W oknie dialogowym Add New Item – Instrumenty na liście Template wybierz szablon Class. Następnie w polu Name wpisz Fabryka. Kliknij Add.
7. Do klasy Fabryka dodaj odpowiednią funkcjonalność	<ul style="list-style-type: none"> Przejdź do pliku Fabryka.cs. Do klasy Fabryka dodaj metodę Utworz: <pre>public static Instrument Utworz(int i) { Instrument instr = null; switch (i) { case 1: instr = new Instrument(); break; case 2: instr = new Trabka(); break; } return instr; }</pre> Do klasy Fabryka dodaj metodę Menu: <pre>public static int Menu() { Console.Clear(); Console.WriteLine("\n\t\t\t1 - Informacje ogólne o instrumencie"); Console.WriteLine("\n\t\t\t2 - Informacje o trąbce"); Console.WriteLine("\n\t\t\t0 - Koniec"); int i; bool b; do { do { b = int.TryParse(Console.ReadLine(), out i); } while (!b); } while (0 > i i > 2); return i; }</pre>
8. Zaimplementuj silnik programu	<ul style="list-style-type: none"> Z menu wybierz Project -> Add Class. W oknie dialogowym Add New Item – Instrumenty na liście Template wybierz szablon Class. Następnie w polu Name wpisz Klient. Kliknij Add.

	<ul style="list-style-type: none"> Przejdź do pliku Klient.cs. Do klasy Klient dodaj pole typu Instrument: <pre>private Instrument instrument;</pre> Do klasy Klient dodaj metodę Menu: <pre>public int Menu() { Console.Clear(); Console.WriteLine("Twój aktualny wybór to {0}\n", instrument); Console.WriteLine("\n\t\t\t\t\t1 - Przykładowe dźwięki " + "wydawane przez instrument"); Console.WriteLine("\n\t\t\t\t\t2 - Krótka charakterystyka " + "instrumentu"); Console.WriteLine("\n\t\t\t\t\t3 - Powrót do menu głównego"); int i; bool b; do { b = int.TryParse(Console.ReadLine(), out i); } while (!b); return i; }</pre> Do klasy Klient dodaj metodę Uruchom: <pre>public void Uruchom() { int i, j; while (true) { i = Fabryka.Menu(); if (i == 0) break; instrument = Fabryka.Utworz(i); do { j = Menu(); Console.Clear(); switch (j) { case 1: instrument.Graj(); Console.ReadKey(); break; case 2: Console.WriteLine(instrument.Opis); Console.ReadKey(); break; } } while (j != 3); } }</pre> Przejdź do pliku Program.cs. Do metody Main dodaj następujący kod: <pre>Klient k = new Klient(); k.Uruchom();</pre>
9. Skompiluj i uruchom program	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Problem 2 (czas realizacji 10 minut)

Szef kazał Ci dodać do edukacyjnego programu muzycznego również informacje o bębnie. Masz napisaną klasę **Beben**. Niestety nie masz dostępu do kodu klasy. Implementację dostałeś w formie

biblioteki DLL. Stosując wzorec projektowy adapter spróbuj rozszerzyć funkcjonalność swojej aplikacji. Klasa **Beben** posiada dwie metody:

- **Uderz** – „imituje” wydawanie dźwięku przez bęben
- **ZwrocOpis** – zwraca krótką charakterystykę tego instrumentu

Pliki wymagane w tym laboratorium znajdują się w katalogu **Kurs\Lab\Start\Modul08**, gdzie **Kurs** jest katalogiem w którym zainstalowano pliki kursu.


Zadanie	Tok postępowania
1. Do bieżącego rozwiązania dodaj referencję do biblioteki Beben.dll	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt, a następnie z menu kontekstowego wybierz Add Reference. • W oknie dialogowym Add Reference przejdź do zakładki Browse. • Przejdź do katalogu Kurs\Lab\Start\Modul08, gdzie Kurs jest katalogiem, w którym zainstalowano pliki kursu. • Zaznacz plik Beben.dll. • Kliknij OK.
2. Utwórz klasę opakowującą	<ul style="list-style-type: none"> • Z menu wybierz Project -> Add Class. • W oknie dialogowym Add New Item – Instrumenty na liście Template wybierz szablon Class. Następnie w polu Name wpisz AdoptowanyBeben. • Kliknij Add. • Przejdź do pliku AdoptowanyBeben.cs. • Zaimportuj przestrzeń nazw InstrumentyPerkusyjne, w której zdefiniowana jest klasa Beben: <pre>using InstrumentyPerkusyjne;</pre> • Zaznacz, że klasa AdoptowanyBeben dziedziczy po klasie Instrument: <pre>class AdoptowanyBeben : Instrument</pre> • Do klasy AdoptowanyBeben dodaj pole, którego typ jest określony przez klasę Beben. Utwórz obiekt w miejscu definicji pola: <pre>private Beben beben = new Beben();</pre> • W klasie AdoptowanyBeben nadpisz metodę Graj, wykorzystując wcześniej zdefiniowane pole beben: <pre>public override void Graj() { Console.WriteLine(beben.Uderz()); }</pre> • W klasie AdoptowanyBeben nadpisz właściwość Opis, wykorzystując wcześniej zdefiniowane pole beben: <pre>public override string Opis { get { return beben.ZwrocOpis(); } }</pre> • W klasie AdoptowanyBeben nadpisz metodę ToString: <pre>public override string ToString() { return "bęben"; }</pre>
3. Zmodyfikuj	<ul style="list-style-type: none"> • Przejdź do pliku Fabryka.cs.


<p>klasę Fabryka, aby program zawierał również informacje o bębnie</p>	<ul style="list-style-type: none"> • Zmodyfikuj metodę Menu, dodając nową możliwość – wybór bębna: <pre>public static int Menu() { Console.Clear(); Console.WriteLine("\n\t\t\t\t1 - Informacje ogólne o instrumencie"); Console.WriteLine("\n\t\t\t\t2 - Informacje o trąbce"); Console.WriteLine("\n\t\t\t\t2 - Informacje o bębnie"); Console.WriteLine("\n\t\t\t\t0 - Koniec"); int i; bool b; do { do { b = int.TryParse(Console.ReadLine(), out i); } while (!b); } while (0 > i i > 3); return i; }</pre> <ul style="list-style-type: none"> • Zmodyfikuj metodę Utworz, dodając nową możliwość – utworzenie obiektu klasy AdoptowanyBeben: <pre>public static Instrument Utworz(int i) { Instrument instr = null; switch (i) { case 1: instr = new Instrument(); break; case 2: instr = new Trąbka(); break; case 3: instr = new AdoptowanyBeben(); break; } return instr; }</pre>
<p>4. Skompiluj i uruchom program</p>	<ul style="list-style-type: none"> • Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. • W celu uruchomienia programu z menu Debug wybierz Start Debugging.


Problem 3 (czas realizacji 15 minut)

Twoim zadaniem jest uaktualnienie klasy **Samochod**. Klasa ma być wykorzystywana do ewidencji i wyszukiwania samochodów w bazie policyjnej. Musisz nadpisać metody **ToString**, **Equals** i **GetHashCode**. Przy nadpisywaniu metody **Equals** zwróć uwagę, że ten sam samochód może mieć różnych właścicieli.

Pliki wymagane w tym laboratorium znajdują się w katalogu **Kurs\Lab\Start\Modul08**, gdzie **Kurs** jest katalogiem w którym zainstalowano pliki kursu.

Zadanie	Tok postępowania
<p>1. Do bieżącego rozwiązania dodaj projekt Samochody</p>	<ul style="list-style-type: none"> • Uruchom eksplorator Windows i przejdź do katalogu Kurs\Lab\Start\Modul08, gdzie Kurs jest katalogiem, w którym zainstalowano pliki kursu. • Skopiuj do katalogu bieżącego rozwiązania katalog Samochody. <p> W celu sprawdzenia nazwy katalogu, w którym zapisane zostało rozwiązanie, w oknie Solution Explorer zaznacz element reprezentujący rozwiązanie. W okienku Properties w polu Path możesz znaleźć ścieżkę do odpowiedniego katalogu.</p>

	<ul style="list-style-type: none"> W oknie Solution Explorer kliknij prawym klawiszem myszy element reprezentujący rozwiązanie i z menu kontekstowego wybierz Add -> Existing Project. W oknie dialogowym Add Existing Project przejdź do katalogu rozwiązania, później do katalogu Samochody, a następnie zaznacz plik Samochody.csproj. Kliknij OK.
2. Dostosuj klasę Samochod do nowych wymagań	<ul style="list-style-type: none"> Przejdź do pliku Samochod.cs. Plik ten znajduje się w projekcie Samochody. Dla klasy Samochod nadpisz metodę klasę ToString. Metoda ta będzie zwracać stan obiektu klasy Samochod: <pre>public override string ToString() { return string.Format("{0} {1} rok produkcji: {2}; pojemność " + "silnika: {3}; numer silnika: {4}; właściciel: {5} {6}", Producent, Marka, RokProdukcji, Silnik.Pojemnosc, Silnik.NumerFabryczny, Wlasciciel.Imie, Wlasciciel.Nazwisko); }</pre> Dla klasy Samochod nadpisz metodę klasę Equals. Przy sprawdzaniu równości nie bierz pod uwagę właściwości Wlasciciel: <pre>public override bool Equals(object obj) { if (obj == null) return false; if (this.GetType() != obj.GetType()) return false; Samochod s = (Samochod)obj; if (RokProdukcji != s.RokProdukcji) return false; if (!string.Equals(Producent, s.Producent)) return false; if (!string.Equals(Marka, s.Marka)) return false; if (!object.Equals(Silnik, s.Silnik)) return false; return true; }</pre> Dla klasy Samochod nadpisz metodę klasę GetHashCode: <pre>public override int GetHashCode() { return string.Format("{0} {1} {2} {3} {4}", Producent, Marka, RokProdukcji, Silnik.NumerFabryczny, Silnik.Pojemnosc).GetHashCode(); }</pre> <p> Dlaczego nie wykorzystujemy metody ToString do uzyskania napisu, który byłby użyty do generacji wartości kodu skrótu?</p>
3. Napisz program testujący zmodyfikowaną klasę Samochod	<ul style="list-style-type: none"> W oknie Solution Explorer kliknij prawym klawiszem myszy element reprezentujący rozwiązanie i z menu kontekstowego wybierz Add -> New Project Z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestSamochodu. Kliknij OK. W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestSamochodu, a następnie z menu kontekstowego wybierz Set as StartUp Project. W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestSamochodu, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects. Zaznacz projekt Samochody.

	<ul style="list-style-type: none"> Kliknij OK. Przejdź do pliku Program.cs. Plik ten znajduje się w projekcie TestSamochodu. Zaimportuj przestrzeń nazw Samochody. <pre>using Samochody;</pre> Do metody Main dodaj następujący kod: <pre>Silnik silnik1 = new Silnik(2.2, 123); Silnik silnik2 = new Silnik(2.2, 123); Osoba osoba1 = new Osoba() { Nazwisko = "Kowalski", Imie = "Jan" }; Osoba osoba2 = new Osoba() { Nazwisko = "Nowak", Imie = "Paweł" }; Samochod s1 = new Samochod(osoba1, "A2", "Audi", 2001, silnik1); Samochod s2 = new Samochod(osoba2, "A2", "Audi", 2001, silnik1); Samochod s3 = new Samochod(osoba1, "A2", "Audi", 2001, silnik2); Console.WriteLine("Samochód s1: {0}", s1); Console.WriteLine("Samochód s2: {0}", s2); Console.WriteLine("Samochód s1: {0}\n", s3); Console.WriteLine("Samochód s1 kod skrótu: {0}", s1.GetHashCode()); Console.WriteLine("Samochód s2 kod skrótu: {0}", s2.GetHashCode()); Console.WriteLine("Samochód s3 kod skrótu: {0}\n", s3.GetHashCode()); Console.WriteLine("s1 i s2 są sobie równe to: {0}", s1.Equals(s2)); Console.WriteLine("s1 i s3 są sobie równe to: {0}", s1.Equals(s3)); Console.WriteLine("s2 i s3 są sobie równe to: {0}", s3.Equals(s2)); Console.ReadKey();</pre>
4. Skompiluj i uruchom program	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. W celu uruchomienia programu, z menu Debug wybierz Start Debugging. <p> Dlaczego obiekty s1 i s2 są sobie równe chociaż mają różny stan – różnią się imieniem i nazwiskiem właściciela samochodu? Dlaczego obiekty s1 i s3 są różne, a mają tę samą wartość kodu skrótu oraz ten sam napis reprezentujący stan obiektu? Czy jest to zgodne z zasadami obowiązującymi w języku C#, aby różne obiekty zwracały ten sam kod skrótu?</p>

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 90 min)

Firma, w której pracujesz, dostała zlecenie na napisanie programu wspierającego pracę biblioteki. Biblioteka, oprócz książek, w swoich zbiorach zawiera również czasopisma, płyty CD i DVD zarówno audio, jak i przechowujące programy komputerowe, a także kasety audio i VHS. O każdym z materiałów bibliotecznych potrzebujesz następujących informacji:

- książka** – imię i nazwisko autora, ew. autorów, tytuł książki, rok wydania, nazwa wydawnictwa, numer ISBN
- czasopismo** – tytuł czasopisma, numer, rok wydania, czy jest to kwartalnik, miesięcznik itp.
- płyta CD/DVD audio** – tytuł płyty, spis wykonawców, spis utworów, typ nośnika (DVD czy CD)
- płyta CD/DVD przechowujące programy komputerowe** – nazwa programu, nazwa firmy która jest właścicielem programu
- kaseta audio** – tytuł kasety, spis wykonawców, spis utworów

- **kaseta VHS** – tytuł filmu, nazwisko reżysera

Ponieważ program jest potrzebny już, zdecydowano, że pierwsza wersja będzie umożliwiać ewidencję tylko książek i czasopism. Funkcjonalność obejmująca płyty DVD i CD oraz kasety zostanie dodana w następnej wersji oprogramowania. Twoim pierwszym zadaniem jest opracowanie projektu programu tak, aby był łatwo rozszerzalny o nowe rodzaje materiałów bibliotecznych. Drugim zadaniem jest implementacja programu, który będzie uwzględniał książki i czasopisma.

Program powinien obejmować następującą funkcjonalność:

- dodawanie dowolnego materiału bibliotecznego do bazy
- zapis zaewidencjonowanych materiałów bibliotecznych do pliku (ewentualnie do bazy danych)
- odczytywanie bazy materiałów bibliotecznych z pliku (ewentualnie do bazy danych)
- wyświetlanie informacji o zaewidencjonowanych materiałach bibliotecznych

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 09

Wersja 2

Klasy abstrakcyjne i interfejsy

Spis treści

Klasy abstrakcyjne i interfejsy	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	9
Porady praktyczne	15
Uwagi dla studenta	16
Dodatkowe źródła informacji	16
Laboratorium podstawowe	18

Informacje o module

Opis modułu

W tym module zapoznasz się z pojęciem klas i metod abstrakcyjnych oraz uzyskasz informacje, jak implementować je w języku C#. Dowiesz się również, co to jest interfejs i do czego służy. Poznasz pojęcie klas i metod zamkniętych. Zobaczysz, jak w języku C# sprawdzić, czy zmienna klasy bazowej zawiera referencję do obiektu, którego typ jest żądaną klasą pochodną. Poznasz również wzorzec projektowy fabryki abstrakcyjnej.

Cel modułu

Celem modułu jest przedstawienie sposobów projektowania oraz tworzenia oprogramowania w oparciu o interfejsy i klasy abstrakcyjne.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- Wiedział, co to są klasy i metody abstrakcyjne oraz potrafił je definiować w języku C#
- rozumiał pojęcie interfejsu
- umiał definiować interfejsy oraz je implementować
- wiedział, co to są klasy i metody zamknięte oraz potrafił je definiować
- znał i umiał stosować operatory konwersji as i is
- umiał i wiedział kiedy wykorzystywać wzorzec projektowy fabryki abstrakcyjnej przy projektowaniu oprogramowania

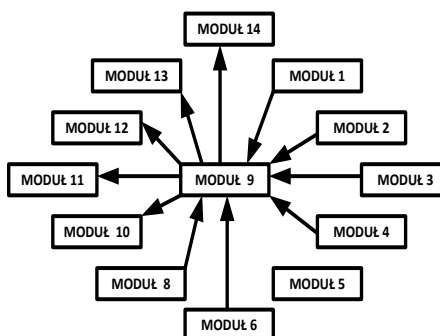
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinienś:

- rozumieć pojęcie dziedziczenia
- rozumieć pojęcie polimorfizmu i metod wirtualnych
- potrafić definiować metody wirtualne i je nadpisywać
- rozumieć wzorzec projektowy metody fabrykującej

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na Rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w module „Pojęcie klasy”, „Konstruktor”, „Właściwości i indeksatory”, „Składowe statyczne”, „Dziedziczenie”, „Polimorfizm i funkcje wirtualne”.



Rys. 13 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Dostałeś od szefa zadanie napisania biblioteki figur geometrycznych. Operacje, które musisz zaimplementować, to między innymi obliczanie pola powierzchni i obwodu figury, rysowanie itp. Dla niektórych figur musisz dodać również możliwość transformacji (skalowanie, translacja, obrót). Oczywiście aby nie powtarzać kodu i móc stworzyć silnik programu w oparciu o zmienną jednej klasy postanowiono, że utworzysz klasę bazową, po której będą dziedziczyć klasy reprezentujące poszczególne figury. I tu zaczyna się problem. Wiemy co to jest trójkąt czy prostokąt, ale jak wygląda figura? Czy może istnieć obiekt takiej klasy? Czy można stworzyć klasę, w której zamknięte będą wspólne cechy i wspólna funkcjonalność, a jednocześnie zapewnić, że nie będzie można utworzyć obiektu tej klasy? Jak przygotować klasę, która będzie pełnić wyłącznie rolę klasy bazowej, a nie szablonu, według którego tworzy się obiekt?

Niektóre klasy nie są ze sobą powiązane zależnością, że jedna jest szczególnym przypadkiem drugiej, a zawierają tę samą grupę funkcji – podobną funkcjonalność. Rysować na ekranie można nie tylko figury, ale również np. zwierzęta. Zarówno klasa reprezentująca trójkąt, jak i klasa reprezentująca kota może zawierać metodę rysującą dany obiekt na ekranie lub zmieniającą jego położenie. Jak definiować wspólną funkcjonalność oraz zaznaczać, że dana klasa zawiera dany zbiór metod?

Wiesz zapewne, że zmienna klasy podstawowej może zawierać referencję do obiektów klas pochodnych. Skoro tak jest, to czy można dostać się do funkcjonalności oferowanej przez klasy pochodne, a nie zawartej w klasie bazowej przy pomocy tej zmiennej? Jak można to zaimplementować w bezpieczny sposób?

W poprzednim module poznałeś metody wirtualne. Dowiedziałeś się również, że wywołanie tych metod składa się niejako z dwóch kroków. To dwuetapowe wywołanie metod wirtualnych powoduje, że są one wolniejsze od zwykłych metod klasy. Mając do napisania bibliotekę figur geometrycznych, w której jednym z zadań będzie obliczanie pola powierzchni, utworzysz zapewne klasę bazową, w której umieścisz metodę wirtualną `ObliczPole`, aby ją nadpisać we wszystkich klasach pochodnych. Jedną z klas może być np. `Kwadrat`. Sposób obliczania pola powierzchni kwadratu nie ulegnie zmianie nawet w przypadku, gdybyś utworzył klasę pochodną `KwadratCzerwony`. Wywołując metodę `ObliczPole`, przy pomocy zmiennej typu `Kwadrat`, kompilator powinien wywołać bezpośrednio metodę `ObliczPole` z klasy `Kwadrat`, ponieważ nigdy nie będzie ona nadpisana. Jak poinstruować kompilator do takiej optymalizacji kodu?

Chcesz znać odpowiedzi na postawione tutaj pytania? Zapoznaj się z treścią tego modułu.

Podstawy teoretyczne

Klasy abstrakcyjne

W programowaniu obiektowym istnieje pojęcie *klasy abstrakcyjnej*. Klasa abstrakcyjna jest to klasa, która nie może mieć swoich przedstawicieli w postaci obiektów. Służy ona jako klasa bazowa w hierarchii klas i zawiera zbiór pól oraz metod wspólnych dla wszystkich klas po niej dziedziczących. W języku C# do definicji klasy abstrakcyjnej używamy słowa `abstract`:

```
[modyfikator_dostępu] abstract class nazwa_klasy_abstrakcyjnej {  
    //definicja klasy abstrakcyjnej  
}
```

Próba utworzenia obiektu, którego typ jest określony przez klasę abstrakcyjną, spowoduje błąd kompilacji. Zmienna tego typu może zawierać jednak referencję do obiektów nieabstrakcyjnych klas pochodnych. Demonstruje to poniższy przykład:

```
abstract class KlasaAbstrakcyjna {  
    //definicja klasy  
}  
  
class KlasaPochodna : KlasaAbstrakcyjna {  
    //definicja klasy  
}  
  
KlasaAbstrakcyjna zmienna1 = new KlasaAbstrakcyjna(); //Błąd kompilacji  
KlasaAbstrakcyjna zmienna2 = new KlasaPochodna();      //OK
```

Jedną z ważnych cech klas abstrakcyjnych jest możliwość definicji wewnątrz nich, oprócz tego, co można definiować wewnątrz zwykłych klas, *metod abstrakcyjnych*. Metoda abstrakcyjna jest to metoda, dla której nie określamy ciała. Może ona występować tylko w klasie abstrakcyjnej. Definiujemy ją w następujący sposób:

```
[modyfikator_dostępu] abstract [typ_zwracany]  
nazwa_metody([lista_parametrów]);
```

Metody abstrakcyjne tworzymy wtedy, gdy w klasie bazowej dla metody nie jesteśmy w stanie dostarczyć sensownej implementacji. Są one wirtualne, ale słowo `virtual` jest niedozwolone obok słowa `abstract`. W klasach pochodnych musimy nadpisać wszystkie metody abstrakcyjne korzystając ze słowa kluczowego `override`, jeżeli klasa pochodna nie jest klasą abstrakcyjną:

```
abstract class KlasaAbstrakcyjna {  
    public abstrakt void Metoda1();  
    public abstrakt void Metoda2();  
}  
  
class KlasaPochodna : KlasaAbstrakcyjna {  
    public override void Metoda1() {  
    }  
}
```

Próba kompilacji powyższego kodu spowoduje błąd. W klasie pochodnej nie dostarczyliśmy implementacji dla metody abstrakcyjnej `Metoda2`. Metoda ta nadal pozostaje więc metodą abstrakcyjną. Klasa pochodna natomiast nie jest klasą abstrakcyjną. Wiemy natomiast, że metody abstrakcyjne mogą być definiowane tylko w klasach abstrakcyjnych.

Metody i klasy zamknięte

W języku C# istnieje pojęcie tzw. *metod zamkniętych*. Metoda zamknięta jest to nadpisana metoda wirtualna, której nie można już nadpisać w klasach pochodnych. Do definicji metod zamkniętych używamy słowa kluczowego `sealed`:

```
class Bazowa {  
    public virtual void Metoda1() {  
    }  
}  
  
class Klasa1 : Bazowa {  
    public sealed override void Metoda1() {  
    }  
}  
  
class Klasa2 : Klasa1 {  
    public override void Metoda1() { //błąd kompilacji  
    }  
}
```

Powyższy kod spowodowałby błąd kompilacji, ponieważ w klasie `Klasa2` nadpisaliśmy metodę zamkniętą z klasy `Klasa1`. Oczywiście pojawia się pytanie, po co stosować ten mechanizm. Spróbujmy to wyjaśnić na następującym przykładzie:

```
class Klasa1 {
    public virtual void Metoda1() {
    }
}

class Klasa2 : Klasa1 {
    public sealed override void Metoda1() {
    }
}

class Test {
    public static void f(Klasa1 b) {
        b.Metoda1();
    }

    public static void g(Klasa2 p) {
        p.Metoda1();
    }
}
```

Przeanalizujmy na początku metodę `f`. W jej ciele wywołujemy metodę `Metoda1` przy pomocy zmiennej typu określonego przez klasę `Klasa1`. Wywoływana metoda jest wirtualna, więc w celu jej wywołania zostanie zastosowane podwójne adresowanie. W czasie kompilacji nie jesteśmy w stanie określić, jaki będzie typ obiektu przesłanego do metody `f` jako argument. W przypadku metody `g`, metodę `Metoda1` wywołujemy przy pomocy zmiennej typu `Klasa2`. W klasie `Klasa2` metoda ta jest zdefiniowana jako zamknięta. Nie może być więc nadpisana w klasach dziedziczących po klasie `Klasa2`. Nie ważne więc jakiego typu (czy klasy `Klasa2`, czy klasy po niej pochodnej) będzie obiekt przesłany jako argument do metody `g`. Zawsze i tak będzie wywołana metoda zdefiniowana w klasie `Klasa2`. Kompilator może więc zastosować bezpośrednio wywołanie metody `Metoda1` z klasy `Klasa2`. Następuje więc optymalizacja kodu.

Słowo `sealed` można stosować nie tylko przy nadpisywaniu metod. Można je również zastosować w przypadku definicji klas, tworząc tzw. *klasę zamkniętą*.

```
[modyfikator_dostępu] sealed class nazwa_klasy_zamknietej {
    //definicja klasy zamkniętej
}
```

Klasa zamknięta jest to klasa, po której nie można dziedziczyć. Metody wirtualne odziedziczone przez tą klasę nie mogą być więc nadpisane – stają się niejawnie metodami zamkniętymi. Wywołanie metody wirtualnej przy pomocy zmiennej, której typ jest określony przez klasę zamkniętą, jest zawsze wywołaniem bezpośrednim.

Przykładem klasy zamkniętej jest typ `string`. Niejawnie typami zamkniętymi są wszystkie struktury.

Interfejs

Interfejs jest to nazwa grupy metod. Z perspektywy interfejsu nie ważna jest implementacja tych metod, ważna jest tylko ich sygnatura (nazwa metody, argumenty metody) oraz typ wartości zwracanej. W języku C# interfejs tworzymy przy pomocy słowa kluczowego `interface`:

```
[modyfikator_dostępu] interface nazwa_interfejsu {
    //deklaracje metod, które tworzą interfejs
}
```

W definicji interfejsu możemy podać:

- deklarację metody

- deklarację właściwości
- deklarację indeksatora
- deklarację zdarzenia – zdarzenia będą dokładnie opisane w module 12 tego kursu

```
interface NazwaInterfejsu {  
    void Metoda1(int arg1);  
    string Wlasciowsc1 {  
        get;  
        set;  
    }  
    double this[int i] {  
        get;  
        set;  
    }  
    event NazwaDelegacji Zdarzenie1;  
}
```

Należy zwrócić uwagę, że żadna z metod deklarowana w definicji interfejsu nie może mieć implementacji. Interfejs nie może zawierać również pól. Przy deklaracji składowych interfejsu nie używamy modyfikatorów dostępu, domyślnie są one publiczne. Nie wolno też użyć innych modyfikatorów: `static`, `virtual`, `override`. Mając zdefiniowany interfejs, możemy zaimplementować go w tworzonych klasach lub strukturach. Implementacja interfejsu polega na zdefiniowaniu wszystkich jego metod w danej klasie lub strukturze. W języku C# do określenia, że klasa implementuje interfejs, stosujemy tę samą składnię jak przy dziedziczeniu:

```
interface IInterfejs1 {  
    void Metoda1();  
    string Metoda2();  
}  
  
class Klasa : IInterfejs1 {  
    public void Metoda1() {  
        //ciało metody  
    }  
    public virtual string Metoda2() {  
    }  
}
```

Implementując metodę interfejsu definiujemy w klasie lub strukturze publiczną metodę o identycznej sygnaturze i typie zwracanym. Metody interfejsu mogą być zaimplementowane jako zwykłe metody. W przypadku klas mogą być również zaimplementowane jako metody wirtualne, gdy planujemy zmienić ich implementację w klasach pochodnych po danej klasie. Warto zwrócić uwagę, że klasa może implementować kilka interfejsów i dziedziczyć po jednej klasie. Nazwa klasy musi występować zawsze jako pierwsza na liście dziedziczenia:

```
class NazwaKlasy : NazwaKlasyBazowej, Interfejs1, Interfejs2 {  
    ...  
}
```

Interfejs może również dziedziczyć po jednym lub więcej interfejsach. Implementując taki interfejs musimy zdefiniować, oprócz metod określonych jawnie przy definicji danego interfejsu, również metody, które zawierają interfejsy, po których nasz interfejs dziedziczy.

Rozważmy przykład, który potraktujmy jako pewną ciekawostkę związaną z implementowaniem interfejsów:

```
class Bazowa {  
    public void Metoda1() {  
    }  
}
```

```
interface Interfejs {  
    public void Metoda1();  
}  
  
class Pochodna : Bazowa, Interfejs {  
}
```

W powyższym przykładzie klasa pochodna implementuje interfejs `Interfejs` przy pomocy metody odziedziczonej z klasy bazowej.

Jawna implementacja interfejsu

Wspomnieliśmy już wcześniej, że jedna klasa może implementować kilka interfejsów. Rozważmy co się stanie, gdy oba interfejsy zawierają identyczną deklarację metody:

```
interface IZolnierz {  
    void Strzelaj();  
}  
  
interface IPilkarz {  
    void Strzelaj();  
}
```

Załóżmy, że implementujemy grę, w której postać może być zarówno piłkarzem jak i żołnierzem.

```
class Postac : IZolnierz, IPilkarz {  
    public void Strzelaj() {  
        //implementacja metody  
    }  
}
```

W powyższym przykładzie pojedyncza metoda implementuje oba interfejsy. W niektórych przypadkach takie podejście będzie poprawne, gdy znaczenie poszczególnych metod z obu interfejsów jest podobne. Jednak co zrobić, gdy każda metoda powinna inaczej działać? W takim przypadku należy co najmniej jeden interfejs zaimplementować w sposób jawny.

```
class Postac : IZolnierz, IPilkarz {  
    public void Strzelaj() {  
        //implementacja metody interfejsu IZolnierz  
    }  
  
    void IPilkarz.Strzelaj() {  
        //jawna implementacja metody Strzelaj interfejsu IPilkarz  
    }  
}
```

Zauważmy, że w przypadku jawnej implementacji metody interfejsu przy definicji metody nie występują żadne modyfikatory. Jawnie zaimplementowaną metodę możemy wywołać tylko przy pomocy zmiennej reprezentującej dany interfejs:

```
Postac postac = new Postac();  
postac.Strzelaj();           //wywołanie metody interfejsu IZolnierz  
IPilkarz pilkarz = postac;  
pilkarz.Strzelaj();          //wywołanie metody interfejsu IPilkarz  
IZolnierz zolnierz = postac;  
zolnierz.Strzelaj();         //wywołanie metody interfejsu IZolnierz
```

Interfejs może być traktowany podobnie jak klasa podstawowa, dlatego zmienne typu określonego przez interfejs mogą zawierać referencję do obiektów klas, które implementują ten interfejs.

Operatory konwersji

Wiemy już, że zmienna klasy bazowej może zawierać referencję do obiektów klas pochodnych. Czasami w programie może zająć potrzeba dokonania konwersji odwrotnej, czyli z klasy bazowej na klasę pochodną lub sprawdzenia, czy zmienna zawiera referencję do obiektu, którego klasa implementuje jakiś interfejs. Konwersja taka nie zawsze może się powieść, dlatego musi być jawnie definiowana.

Pierwszym sposobem konwersji jest użycie nawiasów okrągłych:

```
Klasa1 k1 = (Klasa1) bazowa;
```

W powyższym przykładzie jeżeli zmienna bazowa zawiera referencję do obiektu klasy Klasa1 lub klasy po niej pochodnej, do zmiennej k1 zostanie przypisana referencja do tego obiektu. W przeciwnym wypadku zostanie zgłoszony wyjątek `InvalidCastException`.

Wiadomo, że nie powinniśmy pozostawić nieobsłużonego wyjątku. W celu przechwycenia wyjątku powyższy kod powinien wyglądać następująco

```
try {  
    Klasa1 k1 = (Klasa1) bazowa;  
    //wywołanie metod klasy Klasa1  
}  
catch(InvalidCastException ex) {  
    //obsługa wyjątku  
}
```

Sytuacji wyjątkowych powinno się zawsze unikać. Lepszym rozwiązaniem wydaje się zastosowanie operatora `is`:

```
if(bazowa is Klasa1) {  
    Klasa1 k1 = (Klasa1) bazowa;  
    //wywołanie metod klasy Klasa1  
}
```

Operator `is` zwróci `true` wtedy, gdy zmienna bazowa zawiera referencję do obiektu klasy Klasa1 lub klasy po niej pochodnej. W przeciwnym wypadku zwróci wartość `false`.

Innym sposobem konwersji jest zastosowanie operatora `as`. Operator ten zwróci referencję do obiektu w przypadku, gdy konwertowana zmienna zawiera referencję do obiektu, którego typ jest określony przez żadaną klasę lub klasę po niej pochodną, w przeciwnym wypadku zwróci wartość `null`. Przykład użycia tego operatora jest umieszczony poniżej:

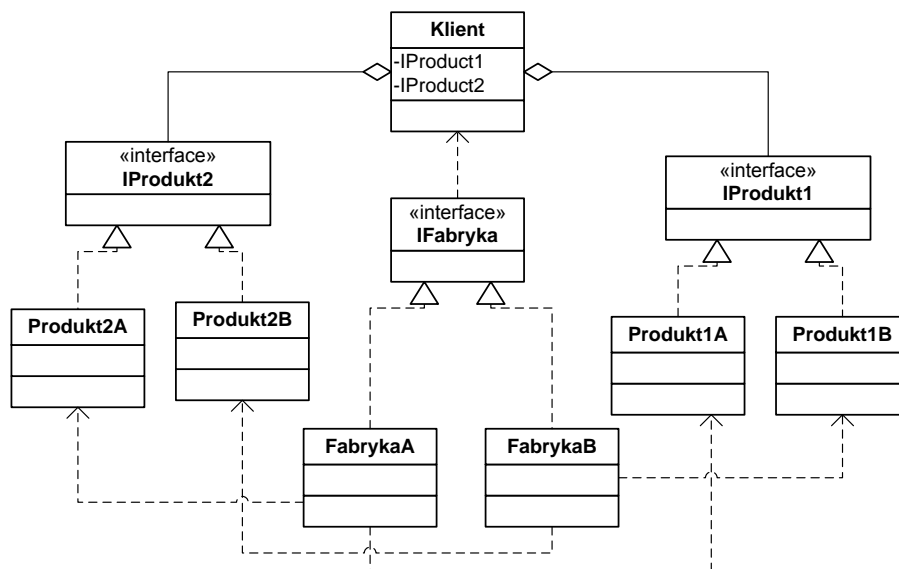
```
Klasa1 k1 = bazowa as Klasa1;  
if(k1 != null) {  
    //wywołanie metod klasy Klasa1  
}
```

Wzorzec projektowy fabryki abstrakcyjnej

W module 8 został opisany wzorzec projektowy metody fabrykującej. Niejako jego rozszerzeniem jest wzorzec *fabryki abstrakcyjnej*, schematycznie przedstawiony na rys. 14.

Wzorzec ten jest wykorzystywany w przypadku, gdy chcemy stworzyć nie pojedynczy produkt, ale grupę powiązanych ze sobą produktów. Klient używa rodziny produktów, oczywiście korzystając ze zmiennych reprezentujących klasy bazowe, klas definiujących konkretne produkty. Można również skorzystać ze zmiennych określonych przez interfejsy (tak jak to ma miejsce na rysunku), które są implementowane przez klasy konkretnych produktów. W odróżnieniu od wzorca metody fabrykującej, klient nie korzysta z konkretnej fabryki, oprócz „abstrakcji” produktów mamy również abstrakcję fabryki. W czasie działania programu, np. w rezultacie wyboru dokonanego przez użytkownika, tworzona jest konkretna fabryka, która tworzy określoną rodzinę produktów. Dodanie nowej rodziny produktów wiąże się tylko z utworzeniem nowej klasy reprezentującej fabrykę wraz z

klasami reprezentującymi nowe produkty, bez konieczności modyfikacji kodu klienta. Przykład zastosowania wzorca projektowego fabryki abstrakcyjnej jest pokazany w dalszej części tego modułu.



Rys. 14 Wzorec projektowy fabryki abstrakcyjnej

Przykładowe rozwiązanie

Tworzenie i wykorzystanie interfejsu

Żałujemy, że chcemy udostępniać stan naszych obiektów na stronach HTML. Musimy zaprojektować odpowiedni interfejs, zawierający metodę zwracającą stan obiektu w postaci HTML, który będzie implementowany w większości nowo tworzonych klas. Program powinien również przewidzieć, że niektóre obiekty, których stan będziemy chcieli umieścić na stronie HTML, będą typu określonego przez klasę, która nie implementuje naszego interfejsu. Możemy założyć, że klasy te są dostarczone przez firmę zewnętrzną i nie mamy dostępu do kodu źródłowego, aby dostosować klasę do nowych potrzeb. Mamy jednak pewność, że we wszystkich tych klasach został nadpisana metoda `ToString`, zwracająca stan obiektu w postaci napisu.

Utworzenie interfejsu

Klasy, które dostarczają funkcjonalność przedstawienia stanu obiektu danej klasy w postaci HTML, będą zaznaczać to przez implementację pewnego interfejsu. Interfejs ten może mieć następującą formę:

```
public interface IHtml {
    string DoHtml();
}
```

Utworzenie klasy implementującej żądany interfejs

Żałujemy, że jedną z klas, której stan obiektu chcemy wyświetlać na stronach WWW jest klasa `Osoba`. Klasa ta została zaimplementowana w następujący sposób.

```
public class Osoba {
    public string Imie { set; get; }
    public string Nazwisko { set; get; }
    public int RokUrodzenia { set; get; }
}
```

Zaznaczmy, że klasa *Osoba* implementuje interfejs *IHtml*, a następnie zaimplementujemy metodę tego interfejsu w danej klasie:

```
public class Osoba : IHtml {
    ...
    public virtual string DoHtmla() {
        return string.Format("<b>Imię: </b><i>{0}</i> <b>nazwisko: " +
            "</b><i>{1}</i> <b>rok urodzenia: </b><i>{2}</i>",
            Imie, Nazwisko, RokUrodzenia);
    }
}
```

Metodę interfejsu w klasie *Osoba* zaimplementowano jako metodę wirtualną, ponieważ w klasach pochodnych po klasie *Osoba* metoda ta może mieć zmienioną funkcjonalność i tak np. dla klasy *Student* kod może wyglądać następująco:

```
public class Student : Osoba {
    public string NumerIndeksu { set; get; }
    public override string DoHtmla() {
        return string.Format("{0} <b>numer indeksu: </b><i>{1}</i>",
            base.DoHtmla(), NumerIndeksu);
    }
}
```

Przypomnijmy, że metody interfejsu nie muszą być implementowane jako metody wirtualne.

Zaimplementowanie metody tworzącej dokument HTML

Metoda tworząca dokument HTML będzie pobierać tablice obiektów typu *object*, których stan chcemy przedstawić na stronach HTML. Dla każdego obiektu z tablicy będziemy sprawdzać, czy jego typ implementuje interfejs *IHtml*. W przypadku gdy interfejs ten jest implementowany, do tworzonego pliku HTML zapisywany jest wynik metody *DoHtmla*. W przeciwnym wypadku do dokumentu HTML zapisywany jest wynik metody *ToString*. Przesłanie do metody tablicy obiektów typu *object* umożliwia przesłanie do metody obiektów (zmiennych) dowolnego typu.

```
static void UtworzDokumentHtml(object[] tab) {
    StreamWriter sw = null;
    try {
        sw = new StreamWriter("test.html");
        sw.WriteLine("<html><head></head><body><ul>");
        foreach (object o in tab) {
            sw.Write("<li>");
            IHtml iHtml = o as IHtml;
            if (iHtml != null) {
                sw.Write(iHtml.DoHtmla());
            }
            else {
                sw.Write(o.ToString());
            }
            sw.WriteLine("</li>");
        }
        sw.Write("</ul></body></html>");
    }
    finally {
        if (sw != null)
            sw.Close();
    }
}
```


Przetestowanie utworzonego kodu

Załóżmy, że oprócz klas *Osoba* i *Student* mamy następującą klasę *Samochod*:

```
public class Samochod {  
    public string Marka { set; get; }  
    public string Producent { set; get; }  
    public override string ToString() {  
        return string.Format("{0} {1}", Producent, Marka);  
    }  
}
```

Zauważmy, że klasa *Samochod* nie implementuje interfejsu *IHtml*. W celu sprawdzenia działania metody *UtworzDokumentHtml* utworzymy tablicę, która będzie zawierać referencję do obiektów typu *Osoba*, *Student* oraz *Samochod* i prześlijmy ją do metody *UtworzDokumentHtml*.

```
static void Main(string[] args) {  
    Osoba osoba = new Osoba() { Imie = "Jan", Nazwisko = "Kowalski",  
        RokUrodzenia = 1990 };  
    Student student = new Student() { Imie = "Paweł", Nazwisko = "Nowak",  
        RokUrodzenia = 1988, NumerIndeksu = "1234" };  
    Samochod samochod = new Samochod() { Producent = "Opel",  
        Marka = "Astra" };  
    object[] obj = { osoba, student, samochod };  
    UtworzDokumentHtml(obj);  
    Console.WriteLine("Dokument HTML został utworzony.");  
    Console.ReadKey();  
}
```

Przykład użycia wzorca projektowego metody abstrakcyjnej

Założmy, że chcemy utworzyć portal, który będzie generował jadłospis dzienny. Użytkownik portalu wybierze sobie rodzaj preferowanej kuchni (jarska, włoska, francuska, dieta), a otrzyma propozycje jadłospisu na śniadanie, obiad i kolację. Jak widać tworzymy zestaw trzech produktów, reprezentujących poszczególne rodzaje posiłków.

Utworzenie klas reprezentujących poszczególne produkty

W rodzinie produktów będą występowały trzy pozycje: śniadanie, obiad i kolacja. Klasy bazowe dla tych produktów zaimplementujemy przy pomocy klas abstrakcyjnych. Dzięki temu zademonstrujemy również użycie klas abstrakcyjnych. W celach demonstracyjnych dodajmy do tych klas pewne pola, aby niejako uzasadnić konieczność utworzenia klasy a nie interfejsu. Implementacja tych klas może wyglądać w następujący sposób:

```
public abstract class SniadanieBase {  
    protected decimal cena;  
    public SniadanieBase(decimal cena) {  
        this.cena = cena;  
    }  
    public abstract string OpisSniadania();  
}  
public abstract class ObiadBase {  
    protected double waga;  
    public ObiadBase(double waga) {  
        this.waga = waga;  
    }  
    public abstract string OpisObiadu();  
}  
public abstract class KolacjaBase {  
    protected double liczbaKalorii;  
    public KolacjaBase(double liczbaKalorii) {  
        this.liczbaKalorii = liczbaKalorii;  
    }  
}
```

```
    }  
    public abstract string OpisKolacji {  
        get;  
    }  
}
```

Dodajmy następnie klasy reprezentujące już konkretne propozycje na poszczególne posiłki, np. kuchni włoskiej:

```
public class SniadanieWloskie : SniadanieBase {  
    public SniadanieWloskie(decimal cena) : base(cena) {  
    }  
    public override string OpisSniadania() {  
        return string.Format("To jest włoskie śniadanie w cenie {0}", cena);  
    }  
}  
  
public class ObiadWloski : ObiadBase {  
    public ObiadWloski(double waga) : base(waga) {  
    }  
    public override string OpisObiadu() {  
        return string.Format("To jest obiad włoski o wadze {0}", waga);  
    }  
}  
  
public class KolacjaWloska : KolacjaBase {  
    public KolacjaWloska(double iloscKalorii) : base(iloscKalorii) {  
    }  
    public override string OpisKolacji {  
        get { return string.Format(  
            "To jest włoska kolacja która zawiera {0} kalorii",  
            loscKalorii); }  
    }  
}
```

Utworzenie klas reprezentujących poszczególne fabryki

Bazowa fabryka będzie zawierać trzy metody: `PrzygotujSniadanie`, `PrzygotujObiad` oraz `PrzygotujKolacje`. Nie posiada ona żadnej implementacji, jest to tylko zbiór metod, zaimplementujemy ją więc przy pomocy interfejsu:

```
public interface IKuchnia {  
    SniadanieBase PrzygotujSniadanie();  
    ObiadBase PrzygotujObiad();  
    KolacjaBase PrzygotujKolacje();  
}
```

Implementacja konkretnej kuchni może wyglądać następująco:

```
public class KuchniaWloska : IKuchnia {  
    public SniadanieBase  
        PrzygotujSniadanie() {  
        return new SniadanieWloskie(15);  
    }  
  
    public ObiadBase PrzygotujObiad() {  
        return new ObiadWloski(300);  
    }  
  
    public KolacjaBase PrzygotujKolacje() {  
        return new KolacjaWloska(1200);  
    }  
}
```

```
    }  
}
```

Utworzenie klienta

Zanim utworzymy głównego klienta, najpierw utworzymy klasę odpowiedzialną za umożliwienie użytkownikowi wyboru odpowiedniej kuchni i utworzenie konkretnej fabryki:

```
static class Menu {  
    public static IKuchnia Utworz(int i) {  
        IKuchnia kuchnia = null;  
        switch (i) {  
            case 1: kuchnia = new KuchniaWloska();  
                break;  
        }  
        return kuchnia;  
    }  
    public static int Wyswietl() {  
        Console.WriteLine("\t\t1 - Kuchnia włoska\n");  
        Console.WriteLine("\t\t0 - Koniec");  
        int i;  
        bool b;  
        do {  
            do {  
                b = int.TryParse(Console.ReadLine(), out i);  
            }  
            while (!b);  
        }  
        while (0 > i || i > 1);  
        return i;  
    }  
}
```

Utwórzmy teraz klasę naszego klienta

```
class Klient {  
    private SniadanieBase sniadanie;  
    private ObiadBase obiad;  
    private KolacjaBase kolacja;  
    private IKuchnia kuchnia;  
  
    public int WyswietlMenu() {  
        Console.WriteLine("1 - Propozycja śniadania");  
        Console.WriteLine("2 - Propozycja obiadu");  
        Console.WriteLine("3 - Propozycja kolacji");  
        Console.WriteLine("4 - Powrót do menu główne");  
        int i;  
        bool b;  
        do {  
            b = int.TryParse(Console.ReadLine(), out i);  
        }  
        while (!b);  
        return i;  
    }  
  
    public void Uruchom() {  
        int i, j;  
        while (true) {  
            i = Menu.Wyswietl();  
            if (i == 0)  
                break;  
        }  
    }  
}
```

```

        kuchnia = Menu.Utworz(i);
        sniadanie = kuchnia.PrzygotujSniadanie();
        obiad = kuchnia.PrzygotujObiad();
        kolacja = kuchnia.PrzygotujKolacje();
        do {
            j = Menu();
            switch (j) {
                case 1: sniadanie.OpisSniadania();
                        Console.ReadKey();
                        break;
                case 2: obiad.OpisObiadu();
                        Console.ReadKey();
                        break;
                case 3: kolacja.OpisKolacji();
                        Console.ReadKey();
                        break;
            }
        } while (j != 4);
    }
}

```

Na koniec zaimplementujmy metodę Main:

```

static void Main(string[] args) {
    Klient k = new Klient();
    k.Uruchom();
}

```

Dodanie nowej rodziny produktów

W celu dodania nowej rodziny produktów musimy utworzyć klasy reprezentujące poszczególne posiłki:

```

public class SniadanieDieta : SniadanieBase {
    public SniadanieDieta(decimal cena) : base(cena) {
    }
    public override string OpisSniadania() {
        return string.Format("Mało nie znaczy tanio, cena {0}", cena);
    }
}

public class ObiadDieta : ObiadBase {
    public ObiadDieta(double waga) : base(waga) {
    }
    public override string OpisObiadu() {
        return string.Format("Zaproś przyjaciela, obiad waży {0}", waga);
    }
}

public class KolacjaDieta : KolacjaBase {
    public KolacjaDieta(double iloscKalorii) : base(iloscKalorii) {
    }
    public override string OpisKolacji {
        get { return string.Format("Kolacje oddaj wrogowi, ilość kalorii {0}",
            iloscKalorii); }
    }
}

```

Następnie dodajmy klasę tworzącą powyższy zestaw produktów.

```
public class KuchniaDiete : IKuchnia {    public SniadanieBase
    PrzygotujSniadanie() {
        return new SniadanieDieta(150);
    }

    public ObiadBase PrzygotujObiad() {
        return new ObiadDieta(30);
    }

    public KolacjaBase PrzygotujKolacje() {
        return new KolacjaDieta(12.4);
    }
}
```

Pozostały nam jeszcze tylko drobne zmiany w klasie Menu.

```
static class Menu {
    public static IKuchnia Utworz(int i) {
        ...
        case 2: kuchnia = new KuchniaDiete();
        break;
        ...
    }

    public static int Wyswietl() {
        ...
        Console.WriteLine("\t\t2 - Kuchnia dla osób pragnących schudnąć\n");
        ...
        do {
            ...
        }
        while (0 > i || i > 2);
        return i;
    }
}
```

Gotowy rozwiązany powyższy przykład znajduje się w katalogu **Demo\Modul09**.

Porady praktyczne

- Pamiętaj, że metody abstrakcyjne są metodami wirtualnymi, więc nie mogą być definiowane jako metody prywatne.
- Składowe, które mogą być składowymi abstrakcyjnymi, to: metody, właściwości i zdarzenia. Zdarzenia zostaną omówione w module 12.
- Składowymi interfejsu mogą być również: metody, właściwości i zdarzenia.
- Do nazw klas abstrakcyjnych dodawaj zawsze przyrostek Base do oznaczenia, że jest to klasa bazowa i nie można tworzyć obiektów typu określonego przez tę klasę.
- Nazwę interfejsu rozpoczynaj zawsze wielką literą I, po której występuje właściwa nazwa interfejsu, również rozpoczynająca się wielką literą. Jest to konwencja powszechnie stosowana.
- Struktury przez Ciebie definiowane mogą również implementować interfejsy.
- Konstruktory w klasach abstrakcyjnych definiuj z modyfikatorem dostępu `protected`. Zaakcentuje to dobitniej, że nie możemy bezpośrednio tworzyć obiektów tych klas.
- Klasa abstrakcyjna może posiadać implementację, czyli pola i metody ze zdefiniowaną funkcjonalnością. Interfejs nie może posiadać żadnej implementacji, czyli nie może zawierać definicji pól oraz metod posiadających zdefiniowaną funkcjonalność.
- Klasa może dziedziczyć bezpośrednio tylko po jednej klasie abstrakcyjnej (klasa abstrakcyjna jest też klasą). Klasa lub struktura może natomiast implementować wiele interfejsów.

- Unikaj stosowania operatora konwersji (nawiasów okrągłych) przy sprawdzeniu, czy zmienna zawiera referencję do obiektu, którego typ jest określony przez żadaną klasę lub klasę po niej pochodną, lub czy ten typ implementuje żądany interfejs. W przypadku gdy konwersja się nie powiedzie, zostanie rzucony wyjątek. Nawet gdy go przechwycisz, to i tak rzucony wyjątek ujemnie wpłynie na wydajność Twojego programu. Lepiej jest zastosować operator `is` lub `as`.
- W przypadku gdy Twoja klasa nie definiuje metody abstrakcyjnej lub metody implementowanego interfejsu, a nie chcesz, aby była to klasa abstrakcyjna, zdefiniuj daną metodę w swojej klasie, a w ciele metody rzuć wyjątek `NotSupportedException`.
- Definicję wszystkich metod implementowanego interfejsu trzymaj w jednym miejscu w klasie, najlepiej zamkniętą w bloku region. Blok ten powinien mieć nazwę sugerującą, jakiego interfejsu implementacja jest w nim umieszczona.
- W bazowych klasach abstrakcyjnych używaj składowych abstrakcyjnych do implementacji cech, które są obligatoryjne dla obiektów klas pochodnych.
- Klasy, po których nie będziesz dziedziczył, definiuj jako klasy zamknięte.
- W klasach zamkniętych nie powinieneś definiować składowych chronionych.
- Unikaj jawnej implementacji interfejsu w typach przez siebie definiowanych.
- Nie modyfikuj wcześniej zdefiniowanego interfejsu, szczególnie gdy udostępniłeś go innym programistom.
- Interfejs może dziedziczyć po jednym lub więcej interfejsach.
- Składowe interfejsu implementuj zawsze jako składowe publiczne.
- Pamiętaj, że składowe interfejsu domyślnie nie są wirtualne. Jeżeli chcesz, aby składowa implementowanego interfejsu była wirtualna, zdefiniuj ją ze słówkiem `virtual`.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- znasz i rozumiesz pojęcie klasy abstrakcyjnej
- potrafisz definiować klasy i metody abstrakcyjne
- wiesz, po co definiuje się klasy i metody zamknięte
- potrafisz definiować metody zamknięte
- znasz i rozumiesz pojęcie interfejsu
- potrafisz zdefiniować interfejs
- rozumiesz pojęcie implementacji interfejsu
- wiesz, po co stosuje się jawną implementację interfejsu
- znasz różnice między interfejsem a klasą abstrakcyjną
- wiesz jak sprawdzić, czy zmienna zawiera referencję do obiektu, którego typ jest określony przez żadaną klasę lub ten typ implementuje żądany interfejs

Dodatkowe źródła informacji

1. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka dla programistów chcących nauczyć się programować w języku C#.

2. Stephen C. Perry, *C# i .NET*, Helion, 2006

Książka w porównaniu z poprzednią pozycją skierowana jest do osób trochę bardziej zaawansowanych. Opisuje C# i .NET Framework w wersji 2.0.

3. Andrew Troelsen, *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition*, Apress, 2007

Książka przeznaczona jest dla bardziej zaawansowanych programistów. Czwarte wydanie tej książki opisuje język C# 3.0 i platformę .NET 3.5.

4. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad, jak powinniśmy pisać swoje programy.

5. Codeguru, <http://www.codeguru.pl>

Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.

6. C Sharp Tutorial, http://www.meshplex.org/wiki/C_Sharp_Tutorial

Internetowy kurs języka C#.

7. C# Corner, <http://www.csharpcorner.com>

Portal poświęcony programowaniu w języku C#.

8. Kurs C#, cz. I, http://www.centrumxp.pl/dotNet/20,1,kategoria,Kurs_C_cz_I.aspx

Przystępny kurs języka C# w języku polskim.

Laboratorium podstawowe

Problem 1 (czas realizacji 45 min)

Dostałeś zadanie zaprojektowania i zaimplementowania biblioteki figur geometrycznych. Każda z figur musi posiadać nazwę w celu jej identyfikacji oraz metodę zwracającą napis, opisującą daną figurę. W przypadku figur, dla których jest to możliwe, powinieneś również umożliwić obliczenie obwodu oraz pola powierzchni. Dodatkowo utwórz program testujący przygotowane figury. Figury, które powinieneś zaimplementować na początku, to:

- odcinek
- koło
- trójkąt
- kwadrat

Zwróć uwagę, że trójkąt oraz kwadrat są wielokątami. Każdy wielokąt posiada takie cechy jak:

- współrzędne wierzchołków wielokąta
- liczba boków
- długość każdego boku
- obwód
- pole powierzchni

Uwaga:

- Przy implementacji klasy reprezentującej kwadrat możesz założyć, że boki kwadratu są równoległe do osi układu współrzędnych.
- Możesz uznać, że punkty leżące na jednej prostej tworzą również trójkąt. Nie musisz sprawdzać, czy punkty są współliniowe.
- Do obliczenia pola trójkąta skorzystaj ze wzoru Herona:



$$P = \sqrt{p(p-a)(p-b)(p-c)}$$



gdzie:


a, b, c – długości boków trójkąta

$p = \frac{1}{2}(a + b + c)$ – połowa długości obwodu

Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition	<ul style="list-style-type: none"> • Otwórz Visual C# 2008 Express Edition. • Z menu wybierz File -> New Project. • Z listy Visual Studio installed templates wybierz Class Library. • W polu Name wpisz Figury. • Kliknij OK. • Z menu wybierz File -> Save Figury. • W polu Location wybierz folder w którym będzie zapisany projekt. • Zaznacz pole wyboru Create directory for solution. • W polu Solution Name wpisz Modul09. • Naciśnij przycisk Save
2. Utwórz abstrakcyjną klasę FiguraPlaska	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na FiguraPlaska.cs. W okienku dialogowym naciśnij przycisk Tak. • Uczyń klasę FiguraPlaska klasą abstrakcyjną: <pre>public abstract class FiguraPlaska { }</pre>

	<ul style="list-style-type: none"> Do klasy FiguraPlaska dodaj automatyczną właściwość Nazwa, chroniony konstruktor inicjalizujący tę właściwość oraz metodę abstrakcyjną Opis: <pre>public abstract class FiguraPlaska { public string Nazwa { set; get; } protected FiguraPlaska(string nazwa) { Nazwa = nazwa; } public abstract string Opis(); }</pre>  Czy klasa FiguraPlaska może być zaimplementowana jako zwykła klasa? Czy przy implementacji figury płaskiej możemy użyć interfejsu zamiast klasy abstrakcyjnej?
3. Dodaj do projektu pomocniczą strukturę Punkt	<ul style="list-style-type: none"> Przed definicją klasy abstrakcyjnej FiguraPlaska, wewnątrz przestrzeni nazw Figury dodaj następujący kod: <pre>public struct Punkt { public double X { set; get; } public double Y { set; get; } public override string ToString() { return string.Format("<{0}; {1}>", X, Y); } }</pre>  Struktura reprezentująca punkt jest już zaimplementowana w bibliotece System.Drawing.dll. Jest to struktura System.Drawing.Point.
4. Dodaj definicję interfejsów IObwod i IPole	<ul style="list-style-type: none"> Przed definicją struktury Punkt wewnątrz przestrzeni nazw Figury dodaj następujący kod: <pre>public interface IPole { double ObliczPole(); } public interface IObwod { double ObliczObwod(); }</pre>
5. Do projektu dodaj definicję abstrakcyjnej klasy reprezentującą wielokąt	<ul style="list-style-type: none"> Po definicji klasy abstrakcyjnej FiguraPlaska wewnątrz przestrzeni nazw Figury dodaj definicję klasy abstrakcyjnej Wielobok. Zaznacz, że klasa Wielobok dziedziczy po klasie FiguraPlaska oraz implementuje interfejsy IPole i IObwod: <pre>public abstract class Wielobok : FiguraPlaska, IPole, IObwod { }</pre> Do klasy Wielobok dodaj pole, które będzie przechowywać współrzędne poszczególnych wierzchołków. Wykorzystaj wcześniej zdefiniowaną strukturę Punkt: <pre>protected Punkt[] punkty;</pre> Do klasy Wielobok dodaj konstruktor: <pre>protected Wielobok(string nazwa, int liczbaWierzchołkow) : base(nazwa) { punkty = new Punkt[liczbaWierzchołkow]; }</pre> Do klasy Wielobok dodaj metodę wirtualną zwracającą ilość wierzchołków (boków) danego wielokąta: <pre>public virtual int LiczbaWierzchołkow {</pre>

	<pre> get { return punkty.Length; } } </pre> <ul style="list-style-type: none"> Do klasy Wielobok dodaj wirtualny indeksator, przy pomocy którego będzie można pobrać i ustawić współrzędne poszczególnych wierzchołków: <pre> public virtual Punkt this[int ktoryWierzcholek] { get { return punkty[ktoryWierzcholek]; } set { punkty[ktoryWierzcholek] = value; } } </pre> Do klasy Wielobok dodaj metodę wirtualną zwracającą długość wybranego boku: <pre> public virtual double WyznaczDlugoscBoku(int ktoryBok) { if (ktoryBok < 0 ktoryBok >= LiczbaWierzchołkow) throw new ArgumentException("Podałeś zły indeks boku wieloleta"); double x1, y1, x2, y2; x1 = this[(ktoryBok + 1) % LiczbaWierzchołkow].X; y1 = this[(ktoryBok + 1) % LiczbaWierzchołkow].Y; x2 = this[ktoryBok % LiczbaWierzchołkow].X; y2 = this[ktoryBok % LiczbaWierzchołkow].Y; return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2)); } </pre> W klasie Wielobok zaimplementuj interfejs IObwod: <pre> public virtual double ObliczObwod() { double suma = 0; for (int i = 0; i < LiczbaWierzchołkow; i++) { suma += WyznaczDlugoscBoku(i); } return suma; } </pre> <p> W celu implementacji metod interfejsu w danej klasie, umieść punkt wstawiania (kursor klawiatury) w nazwie żądanego interfejsu na liście dziedziczenia tej klasy. Pod nazwą interfejsu powinna pojawić się wtedy cienka kreseczka (podobna do opcji autokorekty). Przy pomocy wskaźnika myszy lub skrótu klawiszowego CTRL+ możesz rozwinąć tę kreseczkę do menu kontekstowego. Z menu możemy wybrać Implement interface <nazwa interfejsu>, gdy chcemy zaimplementować metody interfejsu w zwykły sposób lub Explicitly implement interface <nazwa interfejsu>, gdy chcemy zaimplementować metody interfejsu w sposób jawny. Po wybraniu jednej z pozycji menu kontekstowego, Visual Studio wygeneruje szkielety metod określonych przez dany interfejs.</p> <ul style="list-style-type: none"> W klasie Wielobok zaimplementuj interfejs IPole. Zaznacz, że metoda ObliczPole w klasie Wielobok jest metodą abstrakcyjną: <pre> public abstract double ObliczPole(); </pre> <p> Zwróć uwagę, że w klasie Wielobok nie zaimplementowaliśmy metody abstrakcyjnej Opis, odziedziczonej po klasie abstrakcyjnej FiguraPlaska. Nie musimy podawać definicji tej metody, ponieważ klasa Wielobok jest również klasą abstrakcyjną.</p>
6. Do projektu dodaj definicję klasy reprezentującą	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt Figury, a następnie z menu kontekstowego wybierz Add -> Class. W oknie dialogowym Add New Item - Figury z listy Templates wybierz

odcinek	<p>element Class.</p> <ul style="list-style-type: none"> W polu Name wpisz Odcinek. Kliknij OK. Zaznacz, że klasa Odcinek jest publiczna i dziedziczy po klasie FiguraPlaska: <pre>public class Odcinek : FiguraPlaska</pre> <ul style="list-style-type: none"> Do klasy Odcinek dodaj automatyczne właściwości reprezentujące współrzędne początku i końca odcinka: <pre>public Punkt A { set; get; } public Punkt B { set; get; }</pre> <ul style="list-style-type: none"> Do klasy Odcinek dodaj publiczny konstruktor: <pre>public Odcinek(Punkt poczatek, Punkt koniec, string nazwa) : base(nazwa) { A = poczatek; B = koniec; }</pre> <ul style="list-style-type: none"> W klasie Odcinek zaimplementuj metodę abstrakcyjną Opis odziedziczoną po klasie FiguraPlaska: <pre>public override string Opis() { return string.Format("Punkt \"{0}\" o początku {1} i końcu {2}", Nazwa,A,B) ; }</pre> <p> W celu automatycznego wygenerowania implementacji metod abstrakcyjnych wybranej klasy abstrakcyjnej możesz posłużyć się podobną techniką, co w przypadku interfejsów.</p>
7. Do projektu dodaj definicję klasy reprezentującą koło	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt Figury, a następnie z menu kontekstowego wybierz Add -> Class. W oknie dialogowym Add New Item - Figury z listy Templates wybierz element Class. W polu Name wpisz Kolo. Kliknij OK. Zaznacz, że klasa Kolo jest publiczna, dziedziczy po klasie FiguraPlaska oraz implementuje interfejsy IPole i IObwod: <pre>public class Kolo : FiguraPlaska, IObwod, IPole</pre> <ul style="list-style-type: none"> Do klasy Kolo dodaj automatyczną właściwość reprezentującą współrzędne środka koła oraz prywatne pole reprezentujące długość promienia i właściwość umożliwiającą dostęp do tego pola: <pre>public Punkt Srodek { set; get; } private double promien; public double Promien { get { return promien; } set { if (value < 0) throw new ApplicationException("Długość promienia nie może mieć wartości ujemnej"); promien = value; } }</pre> <ul style="list-style-type: none"> Do klasy Kolo dodaj publiczny konstruktor: <pre>public Kolo(Punkt srodek, double promien, string nazwa) : base(nazwa) {</pre>

	<pre>Srodek = srodek; Promien = promien; }</pre> <ul style="list-style-type: none"> W klasie Kolo zaimplementuj metodę abstrakcyjną Opis odziedziczoną po klasie FiguraPlaska: <pre>public override string Opis() { return string.Format("Koło \"{0}\" o środku w punkcie {1} i " + " promieniu {2}", Nazwa,Srodek, Promien) ; }</pre> W klasie Wielobok zaimplementuj interfejs IObwod i IPole: <pre>public double ObliczObwod() { return 2 * Math.PI * promien; } public double ObliczPole() { return Math.PI * promien * promien; }</pre>
8. Do projektu dodaj definicję klasy reprezentującą trójkąt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt Figury, a następnie z menu kontekstowego wybierz Add -> Class W oknie dialogowym Add New Item - Figury z listy Templates wybierz element Class. W polu Name wpisz Trojkat. Kliknij OK. Zaznacz, że klasa Trojkat jest publiczna, dziedziczy po klasie Wielobok: <pre>public class Trojkat : Wielobok</pre> Do klasy Trojkat dodaj dwa publiczne konstruktory: <pre>public Trojkat(string nazwa) : base(nazwa, 3) { } public Trojkat(string nazwa, Punkt [] wierzchołki) : base(nazwa, 3) { for (int i = 0; i < 3; i++) { this[i] = wierzchołki[i]; } }</pre> W klasie Trojkat zaimplementuj metody abstrakcyjne Opis i ObliczPole odziedziczone po klasie Wielobok. W celu poprawienia wydajności zaznacz, że metoda ObliczPole jest metodą zamkniętą: <pre>public sealed override double ObliczPole() { double a = WyznaczDlugoscBoku(0); double b = WyznaczDlugoscBoku(1); double c = WyznaczDlugoscBoku(2); double p = (a+b+c)/2; return Math.Sqrt(p * (p - a) * (p - b) * (p - c)); } public override string Opis() { return string.Format("Trojkąt \"{0}\" o wierzchołkach w " + "punktach: A={1}, B={2}, C={3}", Nazwa, this[0], this[1], this[2]); ; }</pre> W celu poprawienia wydajności w programach korzystających z klasy trójkąt zaznacz, że odziedziczone metody WyznaczDlugoscBoku i

	<p>ObliczObwod są w klasie Trojkat metodami zamkniętymi:</p> <pre>public sealed override double WyznaczDlugoscBoku(int ktoryBok) { return base.WyznaczDlugoscBoku(ktoryBok); } public sealed override double ObliczObwod() { return base.ObliczObwod(); }</pre>
<p>9. Do projektu dodaj definicję klasy reprezentującą kwadrat</p>	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt Figury, a następnie z menu kontekstowego wybierz Add -> Class W oknie dialogowym Add New Item - Figury z listy Templates wybierz element Class. W polu Name wpisz Kwadrat. Kliknij OK. Zaznacz, że klasa Kwadrat jest publiczna i dziedziczy po klasie Wielobok: <pre>public class Kwadrat: Wielobok</pre> Do klasy Kwadrat dodaj prywatne pole reprezentujące długość boku oraz właściwość umożliwiającą dostęp do tego pola: <pre>private double dlugoscBoku; public double DlugoscBoku { get { return dlugoscBoku; } set { if (value < 0) throw new ApplicationException("Długość boku nie może mieć wartości ujemnej"); dlugoscBoku = value; } }</pre> Do klasy Kwadrat dodaj publiczny konstruktor: <pre>public Kwadrat(double dlugoscBoku, Punkt punkt, string nazwa) : base(nazwa, 1) { DlugoscBoku = dlugoscBoku; this[0] = punkt; }</pre> W klasie Kwadrat nadpisz wirtualny indeksator: <pre>public override Punkt this[int ktoryWierzcholek] { get { if (ktoryWierzcholek < 0 ktoryWierzcholek > 3) throw new IndexOutOfRangeException("Zły indeks wierzchołka"); switch (ktoryWierzcholek) { case 1: return new Punkt() { X = punkty[0].X + dlugoscBoku, Y = punkty[0].Y }; case 2: return new Punkt() { X = punkty[0].X + dlugoscBoku, Y = punkty[0].Y + dlugoscBoku }; case 3: return new Punkt { X = punkty[0].X, Y = punkty[0].Y + dlugoscBoku }; } return punkty[0]; } set { if (ktoryWierzcholek != 0) throw new ApplicationException("Dla kwadratu możesz ustawić tylko początkowy wierzchołek "); punkty[0] = value; } }</pre>

	<ul style="list-style-type: none"> W klasie Kwadrat zaimplementuj metody abstrakcyjne Opis i ObliczPole odziedziczone po klasie Wielobok. W celu poprawienia wydajności zaznacz, że metoda ObliczPole jest metodą zamkniętą: <pre>public sealed override double ObliczPole() { return dlugoscBoku * dlugoscBoku; }</pre> <pre>public override string Opis() { return string.Format("Kwadrat \"{0}\" o wierzchołkach w " + "punktach: A={1}, B={2}, C={3}, D={4}", Nazwa, this[0], this[1], this[2], this[3]); }</pre> W klasie Kwadrat nadpisz wirtualne metody WyznaczDlugoscBoku i ObliczObwod oraz właściwość LiczbaWierzchołkow. W celu poprawienia wydajności w programach korzystających z klasy Kwadrat zaznacz, że odziedziczone metody WyznaczDlugoscBoku i ObliczObwod oraz właściwość LiczbaWierzchołkow są zamknięte: <pre>public sealed override double ObliczObwod() { return 4 * dlugoscBoku; }</pre> <pre>public sealed override double WyznaczDlugoscBoku(int ktoryBok) { if (ktoryBok < 0 ktoryBok > 3) throw new IndexOutOfRangeException("Zły indeks boku"); return dlugoscBoku; }</pre> <pre>public sealed override int LiczbaWierzchołkow { get { return 4; } }</pre>
10. Do bieżącego rozwiązania dodaj projekt programu wykonywalnego	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestFigur. Kliknij OK.
11. Zaznacz projekt TestFigur jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestFigur, a następnie z menu kontekstowego wybierz Set as StartUp Project.
12. Do projektu TestFigur dodaj odwołanie do biblioteki Figury	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projektu EksploratorTypow, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt TestFigur, a następnie kliknij przycisk OK.
13. Zaimplementuj metodę Main	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Na górze pliku Program.cs zaimportuj przestrzeń nazw Figury. <pre>using Figury;</pre> Przykładowy kod metody Main jest przedstawiony poniżej <pre>static void Main(string[] args) { Punkt[] dlaTrojkata = { new Punkt { X = 1, Y = 1 }, new Punkt { X = 3, Y = 3 }, new Punkt { X = 6, Y = 1 } }; }</pre>

	<pre> FiguraPlaska[] figury = { new Odcinek(new Punkt{X=12,Y=3}, new Punkt{X=1,Y=23}, "Figura 1"), new Kolo(new Punkt{X=3,Y=4},12,"Figura 2"), new Trojkat("Figura 3", dlaTrojkata), new Kwadrat(3,new Punkt{X=6,Y=6}, "Figura 4") }; Console.WriteLine("Mamy do dyspozycji następujące figury: "); foreach (FiguraPlaska f in figury) { Console.WriteLine(f.Opis()); if (f is IPole) { Console.WriteLine("Pole figury wynosi {0}", ((IPole)f).ObliczPole()); } if (f is IObwod) { Console.WriteLine("Obwód figury wynosi {0}", ((IObwod)f).ObliczObwod()); } Wielobok w = f as Wielobok; if (w != null) { Console.WriteLine("Figura posiada następujące wierzchołki: "); for (int i = 0; i < w.LiczbaWierzchołkow; i++) { Console.WriteLine(" - {0}", w[i]); } Console.WriteLine("Figura posiada następujące " + "długości boków: "); for (int i = 0; i < w.LiczbaWierzchołkow; i++) { Console.WriteLine(" - {0}",w.WyznaczDlugoscBoku(i)); } } Console.WriteLine("-----"); } Console.ReadKey(); } </pre>
14. Skompiluj i uruchom program	<ul style="list-style-type: none"> • Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. • W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Laboratorium rozszerzone

Zadanie (czas realizacji 90 min)

Wraz z kolegami postanowiliście stworzyć grę. Gra polega na poruszaniu się bohaterem po zaczarowanym świecie. Bohater charakteryzuje się takimi parametrami jak: siła ataku, obrona, odporność na czary, siła czarów, ilość magii, szybkość itp. Bohater może poruszać się po świecie przy pomocy niezwykłego pojazdu (w zależności od typu terenu pojazd w różny sposób może wpłynąć na szybkość bohatera). Może znajdować artefakty, broń, zdobywać doświadczenie i umiejętności, które mogą podwyższać lub obniżać poszczególne cechy bohatera. Do bohatera mogą przyłączać się różne postacie, które również posiadają specyficzne umiejętności. Zdecydowaliście, że gracz będzie miał do wyboru kilku bohaterów np.: maga, rycerza, złodzieja itp. Należy zwrócić uwagę, że wybór bohatera determinuje następujące rzeczy:

- wartość początkową poszczególnych parametrów bohatera
- początkowe umiejętności bohatera
- umiejętności, które bohater może nabyć
- pojazdy, którymi bohater może się poruszać
- listę artefaktów, które bohater może zabrać ze sobą
- broń, którą bohater może walczyć
- stworzenia, które mogą przyłączyć się do bohatera

Twoim zadaniem jest:

- zaprojektowanie odpowiedniej hierarchii klas, która może zostać użyta do implementacji gry
- utworzenie klas i ewentualnie interfejsów, zgodnie z projektem
- napisanie tej części gry w której gracz wybiera bohatera i odpowiedni obiekt reprezentujący bohatera jest tworzony

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 10

Wersja 2

Metody i typy generyczne

Spis treści

Metody i typy generyczne	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	7
Porady praktyczne	10
Uwagi dla studenta	11
Dodatkowe źródła informacji	11
Laboratorium podstawowe	13
Laboratorium rozszerzone	19
Zadanie 1 (czas realizacji 15 min)	19
Zadanie 2 (czas realizacji 90 min)	19

Informacje o module

Opis modułu

W tym module zapoznasz się z pojęciem metod i typów generycznych. Dowiesz się, do czego służą i jak definiować je w języku C#. Zobaczysz, jakie ograniczenia można stosować względem typów parametryzujących, jak również po co stosować te ograniczenia. Poznasz również kilka typów generycznych reprezentujących podstawowe struktury danych.

Cel modułu

Celem modułu jest przedstawienie pojęcia metod i typów generycznych oraz możliwości ich wykorzystania w języku C#.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- wiedział, co to są typy i metody generyczne
- potrafił definiować typy i metody generyczne
- znał i umiał stosować ograniczenia względem typu parametryzującego typ lub metodę generyczną
- znał podstawowe typy generyczne reprezentujące najpopularniejsze struktury danych

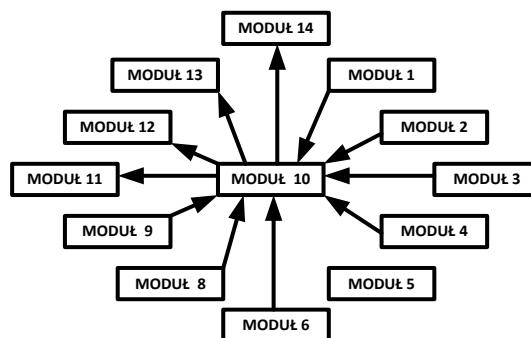
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinienś:

- rozumieć pojęcie klasy
- rozumieć pojęcie metody i potrafić je zdefiniować
- rozumieć pojęcie interfejsu
- wiedzieć, co to jest dziedziczenie
- znać podstawowe struktury danych i algorytmy sortowania

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w modułach „Pojęcie klasy”, „Konstruktor”, „Właściwości i indeksatory”, „Składowe statyczne”, „Dziedziczenie”, „Polimorfizm i funkcje wirtualne” oraz „Klasy abstrakcyjne i interfejsy”.



Rys. 15 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Dostałeś zadanie napisania programu, w którym musisz między innymi wykonać sortowanie liczb rzeczywistych. Po pewnym czasie otrzymałeś zlecenie utworzenia programu, w którym musiałeś sortować obiekty klasy reprezentującej samochód. Sam algorytm sortowania nie zmienił się. Zmienił się tylko typ elementu do sortowania. Metoda sortująca obiekty klasy reprezentującej samochód wykonuje dokładnie te same operacje, co metoda sortująca liczby. Mając zdefiniowaną metodę pozwalającą sortować liczby, wystarczy ją skopiować, zamienić typ reprezentujący liczby rzeczywiste na typ reprezentujący samochód i gotowe.

Sama operacja zamiany typu może być wykonana automatycznie przy pomocy funkcji „znajdź i zamień”, którą posiada praktycznie każdy edytor tekstu. Skoro utworzenie nowej metody operującej na wartościach lub obiektach innego typu może zostać wykonane automatycznie, pojawia się pytanie, czy nie możemy tej operacji zlecić w całości kompilatorowi. Oczywiście, że możemy. Jak to zrobić, dowiesz się czytając ten moduł.

Podobna sytuacja zachodzi przy tworzeniu typów reprezentujących różne struktury danych. Nie ważne, czy lista lub stos zawiera liczby, znaki czy też obiekty Twojej klasy. Struktura będzie wyglądała podobnie i operacje na tej strukturze będą wykonywane w identyczny sposób. Sposób implementacji w języku C# struktur, które mogą być stosowane z wartościami i obiektami różnych typów, również zostanie przedstawiony w tym module.

Podstawy teoretyczne

Metody generyczne

Metody generyczne, zwane również ogólnym lub uniwersalnymi definiujemy w języku C# w następujący sposób:

```
[modyfikator_dostępu] [virtual|static] typ_zwracany
    nazwa_metody<parametry_typu>(argumenty_metody) {
    //definicja metody
}

<parametry_typu> ::= <symbol_typu[,...n]>
```

W odróżnieniu od metod, które do tej pory definiowaliśmy, metody generyczne zawierają umieszczone w nawiasach ostrych parametry typu. Parametry typu są to identyfikatory, które pełnią rolę symbolu zastępczego dla właściwego typu. Brzmi to trochę zawile, więc spróbujemy pokazać to na przykładzie. Rozważmy jeden z bardziej podstawowych algorytmów – algorytm zamiany (ang. *swap*). Z użyciem metod generycznych algorytm ten możemy zaimplementować w następujący sposób:

```
static class MojaKlasa {
    public static void Zamiana<T>(ref T a, ref T b) {
        T tmp = a;
        a = b;
        b = tmp;
    }
}
```

W powyższym przykładzie T pełni rolę nazwy typu. W metodzie możesz za pomocą tego typu definiować argumenty, zmienne lokalne, jak również typ wartości zwracanej. W momencie gdy spróbujemy użyć tej metody, kompilator za T wstawi nazwę konkretnego typu. Przypuśćmy, że chcemy zmienić miejscami dwie zmienne typu `int` oraz dwie zmienne typu `string`:

```
int x = 10, y = 20;
Mojaklasa.Zamiana<int>(ref x, ref y);

string s1 = "Ola", s2 = "Ala";
Mojaklasa.Zamiana(ref s1, ref s2);
```

Podczas kompilacji kompilator automatycznie wygeneruje dwie metody `Zamiana` na podstawie dostarczonego szablonu. W pierwszej metodzie identyfikator `T` zostanie zamieniony na typ `int`, w drugiej zaś – na typ `string`.

Wywołując metodę generyczną możemy podać parametry typu w sposób jawny, co miało miejsce w powyższym przykładzie dla typu `int`. Kompilator może też dopasować typ na podstawie przesłanych argumentów, co ilustruje wywołanie metody z argumentami typu `string`.

Należy zwrócić uwagę, że metoda generyczna nie musi być definiowana jako metoda statyczna.

Klasy generyczne

Klasy generyczne, inaczej ogólne lub uniwersalne definiujemy w języku C# w następujący sposób:

```
[modyfikator_dostępu] class nazwa_klasy<parametry_typu> {
    //definicja klasy
}

<parametry_typu> ::= <symbol_typu[,...n]>
```

Dowolny identyfikator zdefiniowany na liście parametrów typu jest traktowany w całej klasie jako nazwa typu. Klasy generyczne rozważmy na przykładzie definicji kolejki LIFO, czyli stosu. Przy pomocy klas generycznych implementacja stosu mogłaby wyglądać następująco:

```
public class Stos<T> {
    private const int MAX_ROZMIAR_STOSU = 100;
    private T[] stos = new T[MAX_ROZMIAR_STOSU];
    private int indeks = 0;

    public void Wloz(T a) {
        if (indeks >= MAX_ROZMIAR_STOSU)
            throw new InvalidOperationException ("Stos jest pełny");
        stos[indeks] = a;
        indeks++;
    }

    public T Zdejmij() {
        if (indeks <= 0)
            throw new IndexOutOfRangeException("Stos jest pusty");
        indeks--;
        // Uwaga, jest to wersja uproszczona
        return stos[indeks];
    }
}
```

Mając tak zdefiniowaną klasę generyczną możemy utworzyć stos przechowujący wartości lub obiekty wybranego typu:

```
Stos<int> stos1 = new Stos<int>();
Stos<string> stos2 = new Stos<string>();
Stos<string> stos3 = new Stos<string>();
```

Po napotkaniu powyższego kodu kompilator języka pośrednio automatycznie wygeneruje dwie nowe klasy w oparciu o klasę generyczną. W pierwszym przypadku identyfikator typu `T` zostanie zastąpiony typem `int`, w drugim zaś – typem `string`.

Wartość domyślna typu parametryzującego

Definiując typ lub metodę generyczną możemy mieć niekiedy potrzebę zainicjalizowania zmiennej, której typ określony jest przez parametr, wartością domyślną. Przypomnijmy, że wartość ta dla typów referencyjnych to `null`, dla typów liczbowych zero, dla typu logicznego `false`, natomiast dla pozostałych typów bezpośrednich struktura, której wszystkie pola mają wartością domyślną. Do nadania wartości domyślnej dla zmiennej typu parametrycznego w języku C# używamy słowa `default`:

```
class nazwa_klasy<T> {  
    ...  
    T nazwa_zmiennej = default(T);  
}
```

Stosując słowo `default` możemy poprawić błąd, który popełniliśmy przy implementacji metody `Pop` typu generycznego `Stos`. Załóżmy, że na stosie przechowywane są elementy jakiegoś typu referencyjnego. Po zdjęciu obiektu ze stosu w tablicy nadal istnieje element zawierający referencję do tego obiektu. Referencja ta nie pozwoli usunąć obiektu z pamięci przez garbage collector, nawet wtedy, gdy nie jest on dostępny z żadnego innego miejsca w programie. Dlatego po zdjęciu obiektu ze stosu element tablicy reprezentujący ten obiekt powinien być „wyzzerowany”. Poprawiona wersja metody `Pop` została umieszczona poniżej:

```
public T Pop() {  
    if (index <= 0)  
        throw new IndexOutOfRangeException("Stos jest pusty");  
    index--;  
    T e = stos[index];  
    stos[index] = default(T)  
    return e;  
}
```

Restrykcje dla parametrów typu

Rozważmy metodę wyznaczającą maksimum z dwóch obiektów lub zmiennych. Stosując metodę generyczną moglibyśmy utworzyć następujący kod.

```
public static T Max<T>(T a, T b) {  
    if(a > b)  
        return a;  
    return b;  
}
```

Próba kompilacji powyższego kodu zakończy się niepowodzeniem. Zmienne lub obiekty nie wszystkich typów można porównać. Definiując metodę lub klasę generyczną musimy zadeklarować jakie operacje można wykonać na obiektach lub zmiennych definiowanych przy pomocy identyfikatorów z listy parametrów typów. Określamy to przy pomocy słowa kluczowego `where`.

```
class nazwa_klasy<T, U> [:nazwa_klasy_bazowej] where T: ograniczenie_1  
                                                                where U: ograniczenie_2  
{  
    //definicja klasy generycznej  
}
```

W języku C# mamy sześć rodzajów ograniczeń typu parametryzującego. Należą do nich:

1. Wymaganie dotyczące implementowanych interfejsów przez typ parametryzujący. W przypadku, gdy typ parametryzujący ma implementować kilka interfejsów, poszczególne nazwy interfejsów rozdzielamy przecinkiem. Interfejsy mogą być również generyczne:

```
class nazwa_klasy<T, U> where T: interfejs_1, interfejs_2
                                where U: interfejs_3<T>
{ }
```

2. Wymaganie, aby typ parametryzujący był typem bezpośrednim. Realizujemy to przy pomocy słowa `struct`:

```
class nazwa_klasy<T> where T: struct
{ }
```

3. Wymaganie, aby typ parametryzujący był typem referencyjnym. Realizujemy to przy pomocy słowa `class`:

```
class nazwa_klasy<T> where T: class
{ }
```

4. Żądanie, aby typ parametryzujący posiadał konstruktor bezparametrowy. Realizujemy to przy pomocy konstrukcji `new()`:

```
class nazwa_klasy<T> where T: new()
{ }
```

5. Wymaganie, aby typ parametryzujący dziedziczył po pewnej klasie. Dany typ parametryzujący może dziedziczyć tylko po jednej klasie. Klasa, po której dziedziczy dany typ parametryzujący, może być również typem generycznym:

```
class nazwa_klasy<T, U> where T: nazwa_klasy_bazowej_1
                                where U: nazwa_klasy_bazowej_2<T>
{ }
```

6. Wymaganie, aby jeden z typów parametryzujących był typem pochodnym po innym z typów parametryzujących:

```
class nazwa_klasy<T, U> where T: U
{ }
```

Warto zwrócić uwagę, że niektóre typy ograniczeń można ze sobą łączyć.

Zastosowanie ograniczenia powoduje, że dany typ parametryzujący może być zastąpiony tylko typem spełniającym dane ograniczenia. Ma to swoje zalety. Tworząc metodę generyczną lub klasę uniwersalną, korzystając ze zmiennych danego typu parametryzującego możemy korzystać z cech wynikających z ograniczenia. I tak np. jeżeli nasz typ parametryzujący dziedziczy po pewnej klasie, to korzystając ze zmiennej tego typu możemy korzystać ze wszystkich publicznych składowych klasy bazowej.

Wracając do naszego przykładu wyznaczającego maksimum, musimy zaznaczyć, że zmienne typu parametryzującego możemy porównywać. Typy, których zmienne możemy porównywać w .NET Framework implementują interfejs `IComparable`. Więcej na temat interfejsu `IComparable` można przeczytać w rozdziale 11. Prawidłowa implementacja metody wyznaczającej maksimum powinna wyglądać więc następująco:

```
public static T Max<T>(T a, T b) where T: IComparable<T> {
    if(a.CompareTo(b)>0)
        return a;
    return b;
}
```

W powyższym przykładzie, ponieważ typ `T` implementuje interfejs `IComparable`, przy pomocy zmiennych typu `T` możemy wywołać metodę `CompareTo`, która jest składową wspomnianego interfejsu.

Typy generyczne w bibliotece .NET Framework

W bibliotece .NET Framework przy pomocy typów generycznych mamy zaimplementowanych szereg podstawowych struktur danych. Większość tych typów zdefiniowana jest w przestrzeni nazw `System.Collections.Generic`. Należą do nich między innymi:

- `Queue<T>` – kolejka FIFO („pierwszy przyszedł, pierwszy wyszedł”)
- `Stack<T>` – kolejka LIFO (stos)
- `List<T>` – kolekcja elementów, które mogą być dostępne za pomocą indeksu, należy zwrócić uwagę, że wbrew nazwie kolekcja ta jest zaimplementowana na bazie tablicy, a nie listy połączonej
- `LinkedList<T>` – lista dwukierunkowa
- `Dictionary<TKey, TValue>` – słownika, jest również implementowany przy pomocy wewnętrznej tablicy

Więcej na temat poszczególnych struktur danych oraz przykłady użycia można znaleźć w kursie „Wprowadzenie do programowania”.

Przykładowe rozwiązanie

Definicja metody generycznej

Naszym zadaniem jest stworzenie dwóch metod. Pierwsza metoda będzie wypisywać elementy tablicy. Druga będzie sortować tablicę elementów dowolnego typu stosując sortowanie przez wybieranie. Przypomnijmy, że sortowanie przez wybieranie polega na wyszukaniu elementu mającego znaleźć się na zadanej pozycji i zamianie miejscami z tym, który jest tam obecnie.

Utworzenie metody wypisującej elementy tablicy

Metoda przez nas definiowana będzie wykorzystywać metodę `ToString` do uzyskania stanu obiektu w formie napisu. Na typ parametryzowany nie musimy nakładać żadnych ograniczeń, ponieważ każdy typ posiada tę metodę odziedziczoną z typu `object`. Oczywiście jeżeli typ, który użyjemy, nie ma nadpisanej tej metody, zostanie wypisana pełna nazwa typu, a nie stan obiektu:

```
class Program {
    public static void Wypisz<T>(T[] tab) {
        foreach(T a in tab) {
            Console.Write("{0}, ", a.ToString());
        }
        Console.WriteLine();
    }
}
```

Utworzenie metody sortującej

Zwróćmy uwagę, że aby posortować elementy musimy móc je porównać. W tym celu na typ parametryzujący musimy nałożyć ograniczenie. Zaznaczymy, że typ parametryzujący implementuje interfejs `Comparable`. Ponadto użyjemy generycznej wersji tego interfejsu. Dzięki temu nie będziemy musieli wykonywać rzutowania z typu `object` na typ parametryzowany – wersja zwykła (nie generyczna) przyjmuje jako argument zmienną typu `object`. Przykładowa implementacja żądanej metody znajduje się poniżej:

```
public static class MetodySortujace {
    public static void SortowaniePrzezWstawianie<T>(T[] tab)
        where T: Comparable<T> {
        int i, j, k;
        T x;
        for (i = 0; i < tab.Length; i++) {
            k = i;
            for (j = i + 1; j < tab.Length; j++) {
```

```

        if (tab[j].CompareTo(tab[k]) < 0)
            k = j;
    }
    if (k != i) {
        x = tab[i];
        tab[i] = tab[k];
        tab[k] = x;
    }
}
}
}

```

Zastosowanie utworzonych metod generycznych

Mając zdefiniowane następujące tablice:

```

string[] zoo = { "lew", "antylopa", "żyrafa", "wilk", "tygrys" };
int[] numery = { 5, 45, 21, 3, 23, 34, 66, 55 };

```

spróbujemy posortować je oraz wypisać ich elementy. Wykorzystując wcześniej zdefiniowane metody zadanie to można zrealizować w następujący sposób:

```

MetodySortujace.SortowaniePrzezWstawianie(zoo);
MetodySortujace.SortowaniePrzezWstawianie(numery);

Program.Wypisz(zoo);
Program.Wypisz(numery);

```

W wyniku działania powyższego kodu na ekranie otrzymamy:

```

antylopa, lew, tygrys, wilk, żyrafa,
3, 5, 21, 23, 34, 45, 55, 66,

```

Utworzenie klasy generycznej

Załóżmy, że naszym zadaniem jest implementacja listy jednokierunkowej.

Utworzenie klasy reprezentującej listę jednokierunkową

Tworzymy listę, która może przechowywać elementy żadanego typu. Nie chcemy tworzyć listy uporządkowanej, dlatego na typ parametryzujący nie będziemy nakładać żadnych ograniczeń. Sam deklaracja listy może wyglądać w następujący sposób:

```

public class ListaJednokierunkowa<T> {
    ...
}

```

Wewnątrz listy utworzymy klasę reprezentującą jej węzeł. Identyfikator T wewnątrz klasy ListaJednokierunkowa jest traktowany jako nazwa typu również dla klasy reprezentującej węzeł. Znajomość klasy Wezeł nie jest konieczna na zewnątrz klasy ListaJednokierunkowa, dlatego uczynimy ją klasą prywatną.

```

public class ListaJednokierunkowa<T> {
    private class Wezeł {
        public T dane;
        public Wezeł nastepny;
        public Wezeł (T x, Wezeł nastepny) {
            this.nastepny = nastepny;
            dane = x;
        }
    }
}

```

Zaimplementujmy listę jednokierunkową zakładając, że typem przechowywanych elementów jest typ T. Przykładowa implementacja znajduje się poniżej:


```
public class ListaJednokierunkowa<T> {
    ...
    Wezel glowa, ogon;
    public ListaJednokierunkowa() {
        glowa = ogon = null;
    }

    public bool CzyPusta() {
        return glowa == null;
    }

    public void DodajDoGlowy(T x) {
        glowa = new Wezel(x, glowa);
        if (ogon == null)
            ogon = glowa;
    }

    public void DodajDoOgona(T x) {
        if (ogon == null) ogon = glowa = new Wezel(x, null);
        else {
            ogon.nastepny = new Wezel(x, null);
            ogon = ogon.nastepny;
        }
    }

    public T UsunZPoczatku() {
        if (glowa == null)
            throw new InvalidOperationException("Nie można usunąć węzła z pustej listy");
        T x = glowa.dane;
        Wezel tmp = glowa;
        if (glowa == ogon)
            glowa = ogon = null;
        else
            glowa = glowa.nastepny;
        return x;
    }

    int LiczbaElementow {
        get {
            int i;
            Wezel tmp;
            for (i = 0, tmp = glowa; tmp != null; i++, tmp = tmp.nastepny);
            return i;
        }
    }

    public T[] ZamienNaTablice() {
        T[] tab = new T[this.LiczbaElementow];
        Wezel tmp = glowa;
        int i=0;
        while (tmp != null) {
            tab[i] = tmp.dane;
            i++;
            tmp = tmp.nastepny;
        }
        return tab;
    }
}
```

Prawidłowy sposób implementacji przechodzenia po kolekcji zostanie przedstawiony w module 11 tego kursu.

Zastosowanie klasy generycznej

Zastosujemy klasę `ListaJednokierunkowa` z typami `int` i `string` w celu demonstracji użycia typów generycznych:

```
static void Main(string[] args) {
    ListaJednokierunkowa<string> miasta = new ListaJednokierunkowa<string>();
    ListaJednokierunkowa<int> numery = new ListaJednokierunkowa<int>();

    miasta.DodajDoGlowy("Warszawa");
    miasta.DodajDoOgona("Łódź");
    miasta.DodajDoOgona("Kraków");
    miasta.DodajDoOgona("Wrocław");
    miasta.DodajDoOgona("Gdańsk");
    foreach(string s in miasta.ZamienNaTablce()) {
        Console.Write("{0}, ", s);
    }
    Console.WriteLine();

    numery.DodajDoGlowy(22);
    numery.DodajDoOgona(12);
    numery.DodajDoOgona(45);
    foreach(int i in numery.ZamienNaTablce()) {
        Console.Write("{0}, ", i);
    }
    Console.ReadKey();
}
```

W wyniku działania powyższego programu na ekranie otrzymamy:

```
Warszawa, Łódź, Kraków, Wrocław, Gdańsk,
22, 12, 45,
```

Gotowy rozwiązany powyższy przykład znajduje się w katalogu **Demo\Moduł10**. Opis listy jednokierunkowej jak i algorytmu sortowania przez wybieranie można znaleźć między innymi w kursie „Wprowadzenie do programowania”.

Porady praktyczne

- Tworząc nowy typ lub metodę zastanów się, czy nie warto utworzyć go/ją jako typ lub metodę generyczną. Może to w późniejszym czasie zaowocować dużą oszczędnością czasu przy ponownym zastosowaniu danego typu lub metody.
- Do identyfikatorów parametrów typów przy definicji metod i typów generycznych używaj wielkich liter.
- Jako typy generyczne mogą być definiowane: struktury, klasy, interfejsy i delegacje. Delegacje zostaną omówione w module 12 tego kursu.
- Pamiętaj, że typ nie może być jednocześnie typem referencyjnym i bezpośrednim, dlatego ograniczeń `struct` i `class` nie można stosować jednocześnie do tego samego typu parametryzującego metodę lub typ generyczny.
- Klasa może dziedziczyć tylko po jednej klasie, więc nazwa klasy może pojawić się na liście ograniczeń danego parametru typu tylko raz.
- Klasa może implementować wiele interfejsów, więc kilka nazw interfejsów może pojawić się na liście ograniczeń danego parametru typu.
- Ponieważ struktury nie mogą dziedziczyć, jeżeli zastosowałeś ograniczenie `struct`, nie możesz podać nazwy klasy jako ograniczenia tego parametru typu. Struktura może jednak

- implementować interfejsy, więc ograniczenie w postaci nazwy interfejsu i słowa `struct` może dotyczyć tego samego parametru typu.
- Stosując wymóg, że nasz typ parametryzujący dziedziczy po jakiejś klasie, niejawnie określamy go jako typ referencyjny. Określenie tego ponownie przy pomocy ograniczenia `class` nie jest dozwolone.
 - Ograniczenie `struct` oraz `class` powinno być zapisane jako pierwsze na liście ograniczającej dany typ parametryzujący metodę i typ generyczny.
 - Ograniczenie zakładające, że dany typ parametryzujący posiada konstruktor bezargumentowy, `new()`, musi być zapisane jako ostateczne na liście ograniczeń danego parametru typu.
 - Stosuj minimalną ilość ograniczeń, tylko te konieczne, aby móc zaimplementować dany typ lub metodę generyczną. Zastosowanie zbyt wielu ograniczeń może spowodować, że z danym typem lub metodą generyczną nie będziesz mógł w przyszłości użyć pewnych typów.
 - Nie używaj „niegenerycznych” typów reprezentujących struktury danych (kolejka, stos, słownik) z przestrzeni nazw `System.Collections`. W zamian używaj ich odpowiedników z przestrzeni `System.Collections.Generic`. Klasy z przestrzeni nazw `System.Collections` przechowują elementy typu `object`, nie zapewniają kontroli typu wstawianego do kolekcji oraz wymagają wykonywania konwersji przy odwoływaniu się do obiektów w kolekcji. Dodatkowo gdy do kolekcji niegenerycznej dodawana jest zmienna typu bezpośredniego, wykonywana jest niejawnie operacja opakowywania (ang. *boxing*), a podczas pobierania – operacja rozpakowywania (ang. *unboxing*).
 - Używając klas kolekcji opartych na tablicy, takich jak `List`, `Queue` czy `Stack`, pamiętaj, że dodanie elementu do kolekcji może spowodować utworzenie nowej tablicy i skopiowanie istniejących elementów do niej, co może być operacją negatywnie wpływającą na wydajność, szczególnie gdy kolekcja zawiera dużo elementów. Najlepiej utworzyć kolekcję tak, by była zdolna od razu pomieścić wymaganą liczbę elementów – jeżeli jesteś w stanie ją przewidzieć.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- znasz i rozumiesz pojęcie typów i metod generycznych
- potrafisz definiować typy i metody generyczne w języku C#
- wiesz po co definiuje się typy i metody generyczne
- znasz ograniczenia jakie można stosować względem parametrów typu
- wiesz, po co stosować ograniczenia względem parametrów typu
- potrafisz definiować ograniczenia na parametry typu w języku C#
- wiesz, jak nadać wartość domyślną zmiennej określonej przez parametr typu
- umiesz stosować typy i metody generyczne
- znasz typy generyczne z przestrzeni nazw `System.Collection.Generic`

Dodatkowe źródła informacji

1. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka skierowana do programistów chcących nauczyć się programować w języku C#.

2. Andrew Troelsen, *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition*, Apress, 2007

Książka przeznaczona dla bardziej zaawansowanych programistów. Czwarte wydanie opisuje język C# 3.0 i platformę .NET 3.5.

3. Jeffrey Richter, *CLR via C#, Second Edition*, Microsoft Press, 2006

Książka opisuje wspólne środowisko uruchomieniowe (CLR) i Microsoft .NET Framework 2.0, wraz z przykładami w Microsoft Visual C# 2005.

4. Jay Hilyard, Stephen Teilhet, *C#. Receptury. Wydanie II*, Helion, 2006

Książka zawiera zbiór porad, które programistom C# pomogą rozwiązać zadania programistyczne, z jakim spotykają się w codziennej pracy.

5. Codeguru, <http://www.codeguru.pl>

Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.

6. C Sharp Tutorial, http://www.meshplex.org/wiki/C_Sharp_Tutorial

Internetowy kurs języka C#.

7. C# Corner, <http://www.csharpcorner.com>


Portal poświęcony programowaniu w języku C#.


Laboratorium podstawowe

Problem 1 (czas realizacji 20 min)

W firmie postanowiono sprawdzić empirycznie kilka metod sortowania, a nuż będą bardziej wydajne przy pewnych rozkładach danych. Ty otrzymałeś za zadanie napisanie metody używającej algorytmu sortowania przez kopcowanie. Testy wydajnościowe sortowania mają być wykonywane na różnych typach, dlatego powinieneś zaimplementować ten algorytm jako metodę generyczną. Algorytm sortowanie przez kopcowanie opisany jest w kursie „Wprowadzenie do programowania”.

Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition typu Class Library	<ul style="list-style-type: none"> Otwórz Visual C# 2008 Express Edition. Z menu wybierz File -> New Project. Z listy Visual Studio installed templates wybierz Class Library. W polu Name wpisz Sortowanie. Kliknij OK. Z menu wybierz File -> Save Sortowanie. W polu Location wybierz folder w którym będzie zapisany projekt. Zaznacz pole wyboru Create directory for solution. W polu Solution Name wpisz Modul10. Naciśnij przycisk Save
2. Utwórz statyczną klasę SortowanieStogowe	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na SortowanieStogowe.cs. W okienku dialogowym naciśnij przycisk Tak. Uczyń klasę SortowanieStogowe klasą statyczną: <pre>public static class SortowanieStogowe { }</pre>
3. Do klasy SortowanieStogowe dodaj prywatną metodę statyczną przesiewanie	<ul style="list-style-type: none"> Do klasy SortowanieStogowe dodaj prywatną metodę statyczną ustawiającą element na odpowiedniej pozycji. Metoda ta powinna być metodą generyczną: <pre>private static void przesiewanie<T>(T[] tablica, int lewy, int prawy) where T : IComparable<T> { int i = lewy, j = 2 * i + 1; T x = tablica[i]; while (j <= prawy) { if (j < prawy) { if (tablica[j].CompareTo(tablica[j + 1]) < 0) j = j + 1; } if (tablica[j].CompareTo(x) < 0) break; tablica[i] = tablica[j]; i = j; j = 2 * i + 1; } tablica[i] = x; }</pre>
4. Do klasy SortowanieStogowe dodaj publiczną metodę statyczną Sortuj	<ul style="list-style-type: none"> Do klasy SortowanieStogowe dodaj publiczną metodę statyczną Sortuj, wykonującą sortowanie stogowe i korzystającą z metody zdefiniowanej w poprzednim kroku. Metoda ta również powinna być metodą generyczną: <pre>public static void Sortuj<T>(T[] tablica) where T : IComparable<T> { int l = tablica.Length / 2, p = tablica.Length - 1;</pre>

	<pre> T x; while (l > 0) { l--; przesiewanie(tablica, l, p); } while (p > 0) { x = tablica[0]; tablica[0] = tablica[p]; tablica[p] = x; p--; przesiewanie(tablica, 0, p); } } </pre> <p> Zauważ, że w metodzie Sortuj na rzecz zmiennej typ T nie ma wywołania metody CompareTo interfejsu IComparable. Dlaczego więc w metodzie Sortuj na parametr typu T nałożono konieczność implementacji interfejsu IComparable?</p>
5. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestSortowania. Kliknij OK.
6. Zaznacz projekt TestSortowania jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestSortowania, a następnie z menu kontekstowego wybierz Set as StartUp Project.
7. Do projektu TestSortowania dodaj odwołanie do biblioteki Sortowanie	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projektu TestSortowania, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt Sortowanie, a następnie kliknij przycisk OK.
8. Zaimplementuj metodę Main	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Na górze pliku Program.cs zaimportuj przestrzeń nazw Sortowanie: <pre>using Sortowanie;</pre> Przetestuj działanie metody Sortuj dla typów string i double: <pre> static void Main(string[] args) { double[] tab1 = { 22.3, 3.5, 1, 7.8, 6.5 }; string[] imiona = { "Paweł", "Ania", "Karol", "Piotr" }; SortowanieStogowe.Sortuj(tab1); SortowanieStogowe.Sortuj(imiona); Console.Write("Imiona: "); foreach (string s in imiona) { Console.Write("{0}, ", s); } Console.WriteLine("\nLiczby: "); foreach (double x in tab1) { Console.Write("{0}; ", x); } Console.ReadKey(); } </pre>

9. Skompiluj i uruchom program	<ul style="list-style-type: none"> • Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. • W celu uruchomienia programu z menu Debug wybierz Start Debugging.
10. Dodaj własną klasę i przetestuj ją z metodą generyczną Sortuj	<ul style="list-style-type: none"> • Wewnątrz przestrzeni nazw TestSortowania dodaj klasę Osoba, posiadającą właściwości Imie oraz Nazwisko: <pre>class Osoba { public string Imie { get; set; } public string Nazwisko { set; get; } }</pre> • W metodzie Main utwórz tablicę zawierającą obiekty klasy Osoba: <pre>Osoba[] osoby = { new Osoba{Imie="Jan", Nazwisko="Kowalski"}, new Osoba{Imie="Tomasz", Nazwisko="Nowal"}, new Osoba{Imie="Marek", Nazwisko="Markowski"} };</pre> • Posortuj tablicę osoby przy pomocy metody SortowanieStogowe.Sortuj: <pre>SortowanieStogowe.Sortuj<Osoba>(osoby);</pre> • Spróbuj skompilować program. •  Czy udało Ci się skompilować program? Jaki błąd kompilacji został zgłoszony? Czym to jest spowodowane? • Umieść w komentarzu dodany w tym punkcie kod.

Problem 2 (czas realizacji 25 min)

Twoja firma postanowiła zwiększyć wydajność operacji wyszukiwania. Postanowiono więc, że podstawową strukturą kolekcji będzie drzewo binarne. W poszczególnych projektach używane są kolekcje, które przechowują elementy różnych typów, dlatego powinieneś użyć do implementacji drzewa binarnego klasy generycznej. Klasa implementująca drzewo binarne powinna mieć następujące metody:

- metodę zwracającą liczbę elementów w drzewie
- metodę dodającą element do drzewa
- metodę wyszukującą element w drzewie
- metodę przekształcającą drzewo w tablicę

Struktura drzewo binarne opisane jest w kursie „Wprowadzenie do programowania”.

Zadanie	Tok postępowania
1. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. • Z listy Visual Studio installed templates wybierz Class Library. • W polu Name wpisz Drzewa. • Kliknij OK.
2. Utwórz klasę DrzewoBinarne	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na DrzewoBinarne.cs. W okienku dialogowym naciśnij przycisk Tak. • Uczyń klasę DrzewoBinarne klasą generyczną, z jednym parametrem

	<p>typu, który implementuje interfejs Comparable<T>:</p> <pre>public class DrzewoBinarne<T> where T: Comparable<T> { }</pre>
3. Do klasy DrzewoBinarne dodaj prywatną klasę reprezentującą węzeł	<ul style="list-style-type: none"> Do klasy DrzewoBinarne dodaj prywatną klasę reprezentującą węzeł. Klasa ta powinna zawierać referencję do prawego węzła dziecka, referencję do lewego węzła dziecka, pole reprezentujące daną przechowywaną w drzewie oraz konstruktor: <pre>private class Wezel { public Wezel lewy; public Wezel prawy; public T dane; public Wezel(T x, Wezel lewy, Wezel prawy) { this.lewy = lewy; this.prawy = prawy; dane = x; } }</pre>
4. Do klasy DrzewoBinarne dodaj pole reprezentujące korzeń drzewa oraz liczbę jego elementów	<ul style="list-style-type: none"> Do klasy DrzewoBinarne dodaj prywatne pole reprezentujące jego korzeń typu Wezel oraz prywatne pole reprezentujące liczbę jego elementów typu int: <pre>private Wezel korzen=null; private int liczbaElementow = 0;</pre>
5. Do klasy DrzewoBinarne dodaj właściwość LiczbaElementow	<ul style="list-style-type: none"> Do klasy DrzewoBinarne dodaj publiczną właściwość tylko do odczytu LiczbaElementow zwracającą liczbę elementów dodanych do drzewa: <pre>public int LiczbaElementow { get { return liczbaElementow; } }</pre>
6. Do klasy DrzewoBinarne dodaj metodę Wstaw	<ul style="list-style-type: none"> Do klasy DrzewoBinarne dodaj publiczną metodę Wstaw dodającą nowy element do drzewa: <pre>public void Wstaw(T x) { liczbaElementow++; if(korzen == null) { korzen = new Wezel(x, null, null); return; } Wezel tmp = korzen, poprzedni; do { poprzedni = tmp; if(tmp.dane.CompareTo(x) < 0) tmp = tmp.prawy; else tmp = tmp.lewy; } while(tmp != null); if(poprzedni.dane.CompareTo(x) < 0) poprzedni.prawy = new Wezel(x, null, null); else poprzedni.lewy = new Wezel(x, null, null); }</pre>
7. Do klasy DrzewoBinarne	<ul style="list-style-type: none"> Do klasy DrzewoBinarne dodaj publiczną metodę Wyszukaj sprawdzającą, czy dany element jest już dodany do drzewa:

dodaj metodę Wstaw	<pre> public bool Wyszukaj(T x) { Wezel tmp = korzen; while(tmp != null) { if(tmp.dane.CompareTo(x) == 0) { return true; } else { if(x.CompareTo(tmp.dane) < 0) { tmp = tmp.lewy; } else { tmp = tmp.prawy; } } } return false; } </pre>
8. Do klasy DrzewoBinarne dodaj metodę ZamienNaTablice	<ul style="list-style-type: none"> Do klasy DrzewoBinarne dodaj publiczną metodę ZamienNaTablice, która tworzy tablicę i kopiuje do niej po kolei elementy drzewa: <pre> public T[] ZamienNaTablice() { T[] tab = new T[LiczbaElementow]; int n = 0; if (korzen != null) { zamienNaTablice(korzen.lewy, tab, ref n); tab[n] = korzen.dane; n++; zamienNaTablice(korzen.prawy, tab, ref n); } return tab; } private static void zamienNaTablice(Wezel w, T[] tab, ref int n) { if (w != null) { zamienNaTablice(w.lewy, tab, ref n); tab[n] = w.dane; n++; zamienNaTablice(w.prawy, tab, ref n); } } </pre>
9. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestDrzewa. Kliknij OK.
10. Zaznacz projekt TestDrzewa jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestDrzewa, a następnie z menu kontekstowego wybierz Set as StartUp Project.
11. Do projektu TestDrzewa dodaj odwołanie do biblioteki Sortowanie	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projektu TestDrzewa, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt Drzewa, a następnie kliknij przycisk OK.
12. Zaimplementuj metodę Main	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Na górze pliku Program.cs zaimportuj przestrzeń nazw Drzewa: using Drzewa;

	<ul style="list-style-type: none"> Przetestuj działanie struktury Sortuj dla typu string i int: <pre> static void Main(string[] args) { DrzewoBinarne<string> imiona = new DrzewoBinarne<string>(); imiona.Wstaw("Wojtek"); imiona.Wstaw("Zosia"); imiona.Wstaw("Piotr"); imiona.Wstaw("Karol"); imiona.Wstaw("Anna"); foreach (string s in imiona.ZamienNaTablice()) { Console.Write("{0}, ", s); } Console.WriteLine("\nPodaj imie: "); string im = Console.ReadLine(); if (imiona.Wyszukaj(im)) { Console.WriteLine("Podane imie znajduje się na liście"); } else { Console.WriteLine("Podanego imienia nie ma na liście"); } DrzewoBinarne<int> numery = new DrzewoBinarne<int>(); numery.Wstaw(11); numery.Wstaw(6); numery.Wstaw(21); numery.Wstaw(55); numery.Wstaw(14); numery.Wstaw(9); foreach (int i in numery.ZamienNaTablice()) { Console.Write("{0}, ", i); } Console.WriteLine("\nPodaj numer: "); int num = int.Parse(Console.ReadLine()); if (numery.Wyszukaj(num)) { Console.WriteLine("Podany numer znajduje się na liście"); } else { Console.WriteLine("Podanego numeru nie ma na liście"); } Console.ReadKey(); } </pre>
13. Skompiluj i uruchom program	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 15 min)

W programach, które tworzy firma, w której pracujesz, konieczne jest definiowanie klas, które będą miały tylko jednego przedstawiciela. Szef firmy wpadł na pomysł, że to właśnie Ty utworzysz klasę generyczną, która będzie implementować wzorzec projektowy singleton. Przypomnijmy, że celem tego wzorca jest właśnie ograniczenie możliwości tworzenia obiektów danej klasy tylko do jednej instancji oraz zapewnienie dostępu do stworzonego obiektu. Wzorzec singleton opisany jest dokładnie w module 4 tego kursu.

Zadanie 2 (czas realizacji 90 min)

W celu poprawienia wydajności wyszukiwania, dane przechowujesz w drzewie binarnym. Niestety dane, które wstawiasz do drzewa, są często posortowane, stąd drzewo ma często postać listy. Postanowiłeś więc utworzyć klasę, która reprezentowałaby drzewo AVL. Ponieważ dane, które przechowujesz i wyszukujesz w poszczególnych projektach są różnych typów, klasę, która reprezentuje drzewo AVL musisz zaimplementować jako klasę generyczną. Struktura drzewa AVL opisana jest w kursie „Wprowadzenie do programowania”.

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 11

Wersja 2

Implementacja przykładowych interfejsów

Spis treści

Implementacja przykładowych interfejsów	1
Informacje o module	2
Przygotowanie teoretyczne	3
Przykładowy problem	3
Podstawy teoretyczne	3
Przykładowe rozwiązanie	8
Porady praktyczne	13
Uwagi dla studenta	13
Dodatkowe źródła informacji	14
Laboratorium podstawowe	15
Problem 1 (czas realizacji 35 min)	15
Problem 2 (czas realizacji 20 min)	18
Laboratorium rozszerzone	22
Zadanie 1 (czas realizacji 90 min)	22
Zadanie 2 (czas realizacji 60 min)	22

Informacje o module

Opis modułu

W tym module zapoznasz się z podstawowymi interfejsami, z których można skorzystać tworząc programy na platformę .NET: IDisposable, IEnumerable, IEnumerator, IComparable i IComparer. Nauczysz się, jak można je zaimplementować we własnych klasach. Dowiesz się, do czego służy blok instrukcji using. Poznasz wzorzec projektowy Iterator oraz składnię iteratorów, przy pomocy których w łatwy sposób można ten wzorzec projektowy zaimplementować.

Cel modułu

Celem modułu jest przedstawienie podstawowych interfejsów biblioteki .NET Framework i sposobów ich implementacji we własnych klasach.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- znał podstawowe interfejsy zdefiniowane w bibliotece FCL
- wiedział do czego służą interfejsy IDisposable, IEnumerable, IEnumerator, IComparable i IComparer
- potrafił implementować powyższe interfejsy we własnych klasach
- umiał używać bloku instrukcji using
- umiał korzystać z konstrukcji iteratorów: yield return oraz yield break

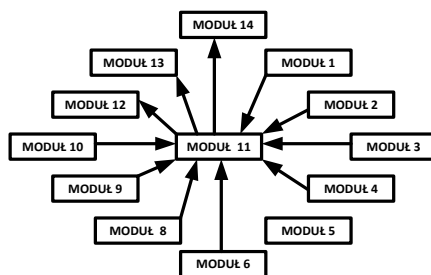
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinienś:

- rozumieć pojęcie interfejsu
- wiedzieć na czym polega implementacja interfejsu
- wiedzieć, co to są typy i metody generyczne
- potrafić definiować własne typy i metody generyczne
- potrafić stosować typy i metody generyczne
- wiedzieć, co to są wzorce projektowe

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w modułach „Pojęcie klasy”, „Właściwości i indeksatory”, „Składowe statyczne”, „Dziedziczenie”, „Polimorfizm i funkcje wirtualne”, „Klasy abstrakcyjne i interfejsy” oraz „Metody i typy generyczne”,



Rys. 16 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Zapewne nie raz w swoim życiu jechałeś samochodem. Umiejąc kierować jednym modelem samochodu jesteś w stanie poprowadzić inny typ samochodu. Dlaczego? We wszystkich typach samochodów, pomijając te z automatyczną skrzynią biegów, z lewej strony masz sprzęgło, w środku hamulec, a z prawej strony pedał gazu. Dobrze by było, aby tak projektować swoje klasy, aby programiści nie musieli się uczyć z nich korzystać, a w celu ich użycia bazowali tylko na swoim doświadczeniu, jak to ma miejsce w przypadku samochodu. Pewnie zastanawiasz się teraz, czy jest to możliwe. Odpowiedź brzmi: tak, pod warunkiem, że Twoje klasy implementują standardowe interfejsy. Tworząc swoje programy korzystasz z klas dostarczonych wraz z biblioteką BCL. Klasy te implementują szereg interfejsów. Używając tych klas, nie zawsze świadomie, korzystałeś z metod różnych interfejsów zaimplementowanych w tych klasach. Jeśli chcesz poznać najważniejsze interfejsy dostępne w bibliotece .NET Framework i dowiedzieć się, jak je implementować we własnych klasach i strukturach, powinieneś przeczytać ten moduł.

Podstawy teoretyczne

Interfejs IDisposable

Wiadomo, że w .NET Framework istnieje mechanizm automatycznego zarządzania pamięcią. Istnieją jednak sytuacje, gdy potrzebujemy zwolnić inne zasoby przydzielone obiektowi naszej klasy przed zwolnieniem pamięci zajmowanej przez obiekt, np. zamknąć połączenie z bazą danych, zamknąć plik itp. W tym celu możemy zdefiniować *finalizator*. Finalizator posiada taką samą nazwę jak klasa, dla której jest definiowany, poprzedzoną znakiem tyldy.

```
class Klasa
{
    ...
    ~Klasa() {
        //definicja finalizatora
    }
}
```

Finalizator nie posiada wartości zwracanej ani parametrów, nie może też być jawnie wywoływany. Wywołanie finalizatora następuje automatycznie w momencie czyszczenia pamięci. Można co prawda wymusić z poziomu programu rozpoczęcie oczyszczania pamięci przy pomocy klasy GC i jej metody `Collect`, ale nie jest to działanie zalecane. Nawet wymuszając proces czyszczenia pamięci nie jesteśmy w stanie przewidzieć, w jakiej kolejności będą wywoływane finalizatory poszczególnych obiektów. Co więcej, finalizator może w ogóle nie zostać wywołany. Podczas zamykania aplikacji wszystkie obiekty przez nią wykorzystywane są usuwane z pamięci. W celu przyspieszenia tego procesu garbage collector może nie wywołać finalizatora dla niektórych obiektów, np. tworzonych w momencie zamykania aplikacji. Podsumowując, finalizator jest metodą wywoływaną niedeterministycznie, więc nie można przewidzieć, jak długo niepotrzebne już dla nas zasoby, np. połączenie z bazą danych, pozostaną zarezerwowane przez nasz obiekt. Chcąc dostarczyć programistom deterministyczny sposób zwalniania zasobów zarezerwowanych przez obiekty naszej klasy, powinniśmy zaimplementować interfejs `IDisposable`. Interfejs ten posiada pojedynczą metodę – `Dispose`. Implementując ten interfejs możemy dostarczyć dla naszej klasy również finalizator. Zabezpieczymy się w ten sposób przed roztargnionymi programistami, którzy zapomnieli zwolnić zasoby ręcznie. Ogólny schemat implementacji interfejsu `IDisposable` i finalizatora został umieszczony poniżej:

```
public class Klasa : IDisposable {
    protected bool disposed = false;
```

```

protected virtual void Dispose(bool disposing) {

    if(disposed)
        return ; //obiekt był już "disposed"
    if(disposing){
        //zwalniamy obiekty, które posiadają meodę Dispose i są
        //wykorzystywane przez obiekt naszej klasy
    }
    // Tutaj zwalniamy zasoby niezarządzane - zarezerwowane przez kod
    //niezarządzany
    disposed = true;
}

public void Dispose() {      //metoda interfejsu IDisposable
    Dispose(true);
    GC.SuppressFinalize(this);
}

~Klasa() {
    Dispose(false);
}
}

```

W klasach pochodnych implementacja ta powinna wyglądać następująco:

```

public class KlasaPochodna : Klasa {
    protected override void Dispose(bool disposing) {
        if(disposed)
            return;
        if(disposing){
            //zwalniamy obiekty, które posiadają meodę Dispose i są
            //wykorzystywane przez obiekt klasy pochodnej
        }
        // Tutaj zwalniamy zasoby niezarządzane - zarezerwowane przez kod
        //niezarządzany, wykorzystywane w klasie pochodnej
        base.Dispose(disposing);
    }
}

```

Wywołanie metody `GC.SuppressFinalize(this)` niejako informuje garbage collector, że zasoby zarezerwowane dla obiektu zostały już zwolnione i nie trzeba dla niego wywoływać finalizatora. Dzięki temu garbage collector podczas procesu oczyszczania pamięci może od razu zwolnić pamięć zajmowaną przez nasz obiekt.

Należy zwrócić uwagę, że kompilator języka C# przekształca finalizator w metodę nadpisującą metodę `Object.Finalize`:

```

protected override void Finalize() {
    try {
        //ciało finalizatora
    }
    finally {
        base.Finalize();
    }
}

```

Konstrukcja using

W języku C# istnieje sposób zautomatyzowania wywołania metody `Dispose`. Realizowane jest to przez blok `using`. Przykład użycia bloku `using` został umieszczony poniżej:

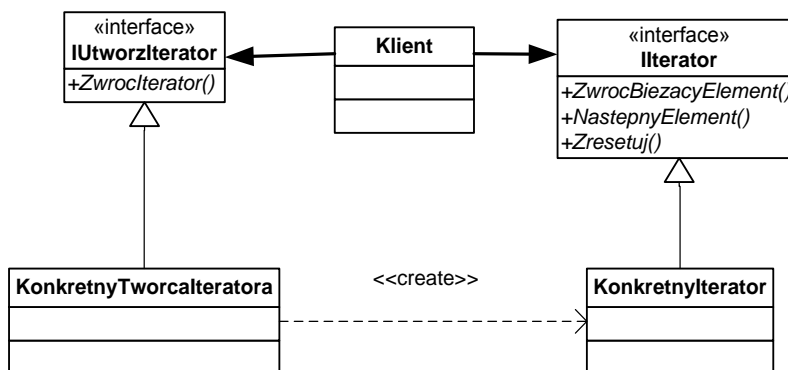

```
using(MojaKlasa k = new MojaKlasa())
{
    //użycie obiektu k
}
```

Klasa `MojaKlasa` musi implementować interfejs `IDisposable`, w przeciwnym wypadku zgłoszony zostanie błąd kompilacji. Powyższy kod jest równoważny następującemu:

```
MojaKlasa k = new MojaKlasa();
try {
    //użycie obiektu k
}
finally {
    k.Dispose()
}
```

Wzorzec projektowy Iterator

Celem wzorca projektowego Iterator jest zapewnienie sekwencyjnego dostępu do elementów kolekcji w taki sposób, aby struktura obiektu reprezentującego tę kolekcję była nieznana dla klienta. Diagram UML przedstawiający schematycznie ten wzorzec jest przedstawiony na rys. 17.



Rys. 17 Wzorzec projektowy iterator

W powyższym diagramie możemy wyróżnić następujące elementy:

- `IUt看orzIterator` – definiuje interfejs służący do tworzenia iteratora
- `IIterator` – definiuje interfejs dostępu sekwencyjnego do elementów kolekcji
- `KonkretnyTworcaIteratora` – tworzy konkretny iterator
- `KonkretnyIterator` – implementuje interfejs `IIterator`, czyli reprezentuje konkretny sposób poruszania się po elementach kolekcji

Interfejsy `IEnumerable` i `IEnumerator`

Przy pomocy interfejsów `IEnumerable` i `IEnumerator` możemy w języku C# zaimplementować wzorzec projektowy iterator. Innymi słowy dzięki zastosowaniu wspomnianych interfejsów, elementy kolekcji – niezależnie od jej rodzaju – mogą być przetwarzane sekwencyjnie w zunifikowany sposób. Interfejs `IEnumerator` posiada trzy składowe:

- Metoda `MoveNext` – wymusza przejście do następnego elementu kolekcji. Zwraca wartość typu `bool`, która informuje, czy udało się przejść do następnego elementu, tj. czy następny element istnieje.
- Metoda `Reset` – ustawia iterator w początkowej pozycji, czyli przed pierwszym elementem.
- Właściwość `Current` – zwraca bieżący element kolekcji.

Istnieją dwie wersje interfejsu `IEnumerator`:

- `System.Collection.IEnumerator` – wersja nietypizowana.

- `System.Collection.Generic.IEnumerator<T>` - wersja generyczna. Interfejs ten w celu zapewnienia zgodności wstecz dziedziczy po wersji nietypizowanej. W tym wariantcie została dodana właściwość `Current`, której typ określony jest przez parametr typu `T`. W przypadku wersji nietypizowanej typem tym był `object`.

Należy zwrócić uwagę, że interfejs `IEnumerator<T>` dziedziczy również po interfejsie `IDisposable`.

Interfejs `IEnumerable` posiada tylko jedną metodę – `GetEnumerator`. Jej rola sprowadza się do zwrócenia obiektu implementującego interfejs `IEnumerator`. W przypadku interfejsu `IEnumerable` również mamy do dyspozycji dwie wersje: nietypizowaną i generyczną. Wersja nietypizowana jako typ zwracany ma nietypizowaną wersję interfejsu `IEnumerator`. Wersja generyczna dziedziczy po wersji nietypizowanej i posiada dodatkowo metodę `GetEnumerator`, której typ wartości zwracanej jest określony przez generyczną odmianę interfejsu `IEnumerator`.

Tworząc klasę, która reprezentuje kolekcję, powinniśmy zaimplementować w niej interfejs `IEnumerable` i zdefiniować oddzielnie klasę implementującą interfejs `IEnumerator`, która umożliwi poruszanie się w określonym porządku po elementach naszej kolekcji. Dla jednej klasy kolekcji możemy mieć zdefiniowanych kilka porządków poruszania się po jej elementach, czyli zdefiniowanych kilka klas implementujących interfejs `IEnumerator`. Szczegóły implementacji tych interfejsów są pokazane w sekcji „Przykładowe rozwiązanie” tego modułu.

W wersji 2.0 języka C# wprowadzono nową składnię tzw. iteratorów. Ta nowa składnia zawiera nowe konstrukcje takie jak *yield return* oraz *yield break*, które bardzo upraszczają implementację wzorca projektowego `Iterator`. Dzięki użyciu tych konstrukcji kompilator automatycznie generuje kod klasy implementującej interfejs `IEnumerator`. Szczegóły użycia składni iteratorów są również pokazane w rozdziale „Przykładowe rozwiązanie” tego modułu.

W przypadku, gdy klasa reprezentująca kolekcję implementuje interfejs `IEnumerable<T>` lub posiada metodę `GetEnumerator`, której typ zwracany jest określony przez interfejs `IEnumerator<T>`, do przetworzenia jej elementów można użyć pętli `foreach`, np.:

```
foreach(TypElementu element in kolekcja)
{
    Console.WriteLine(element);
}
```

Na podstawie kodu przedstawionego powyżej kompilator wygeneruje następujący kod:

```
TypElementu element;
IEnumerator<TypElementu> ie =
    ((IEnumerator< TypElementu >)kolekcja).GetEnumerator();
try {
    while(ie.MoveNext()) {
        element = ie.Current;
        Console.WriteLine(element);
    }
}
finally {
    if(ie is IDisposable)
        ((IDisposable)ie).Dispose();
}
```

Należy zaznaczyć, że zmienna reprezentująca aktualnie przetwarzany element w pętli `foreach` jest tylko do odczytu. Nie można zmienić jej wartości, można natomiast zmodyfikować obiekt, do którego ta zmienna zawiera referencję – w przypadku, gdy typ elementu kolekcji jest typem referencyjnym.

Interfejsy IComparable i IComparer

W celu porównania obiektów naszej klasy możemy przeciążyć operatory relacyjne, jednak lepszym sposobem jest zaimplementowanie interfejsu `IComparable`. Główną zaletą tego podejścia jest to, że metody interfejsu można zaimplementować jako wirtualne. Interfejs ten posiada jedną metodę o nazwie `CompareTo`. Metoda ta zwraca wartość całkowitą. Wartość ujemna oznacza, że obiekt, na rzecz którego wywołano tę metodę jest mniejszy od obiektu przesłanego jako argument. Wartość zero oznacza, że obiekty są sobie równe. Wartość dodatnia pokazuje, że obiekt, na rzecz którego wywołano tę metodę jest większy od obiektu przesłanego jako argument. Sposób implementacji tego interfejsu pokazano poniżej:

```
class Klasa : IComparable<Klasa> {  
    ...  
    private double pole1, pole2;  
    public int CompareTo(Klasa k) {  
        return pole1 - k.pole1;  
    }  
}
```

Przy pomocy interfejsu `IComparer` możemy dostarczyć alternatywne sposoby porównania obiektów naszej klasy. Interfejs ten posiada metodę `Compare`, której działanie jest analogiczne do metody `CompareTo` interfejsu `IComparable`. Przykład implementacji tego interfejsu jest umieszczony poniżej.

```
class Klasa : IComparable<Klasa> {  
    ...  
    private double pole1, pole2;  
    public class PoPolu2 : IComparer<Klasa> {  
        public int Compare(Klasa k1, Klasa k2) {  
            return k1.pole2 - k2.pole2;  
        }  
    }  
}
```

Zdefiniowanie klasy implementującej interfejs `IComparer` jako klasy wewnętrznej umożliwia jej metodom dostęp do pól prywatnych klasy zewnętrznej.

Jeżeli klasa implementuje interfejs `IComparable` lub ma zdefiniowany dla siebie interfejs `IComparer`, tablice obiektów tej klasy możemy sortować przy pomocy metody `Sort` klasy `Array`.

```
Klasa[] tab = {...};  
Array.Sort(tab); //posortowanie obiektów tablic  
//według pola pole1  
Array.Sort(tab, new Klasa.PoPolu2()); //posortowanie obiektów tablic według  
//pola pole2
```

Inne interfejsy

W bibliotece .NET Framework zostało zdefiniowane szereg interfejsów. Oprócz wymienionych wcześniej, należy odnotować następujące interfejsy:

- `IEquatable<T>` – posiada tylko pojedynczą metodę `Equals`, która służy do sprawdzenia równości dwóch obiektów. W odróżnieniu od metody `Equals` odziedziczonej po typie `object`, jako parametr przyjmuje zmienne konkretnego typu. Wewnątrz metody nie musimy więc dokonywać sprawdzenia typu argumentu.
- `IEqualityComparer<T>` – również służy do sprawdzenia równości dwóch obiektów. W odróżnieniu od interfejsu `IEquatable<T>`, oprócz metody `Equals` posiada metodę `GetHashCode`, która zwraca kod skrótu danego obiektu.
- `ICloneable` – posiada metodę `Clone`, która służy do tworzenia kopii obiektu. Więcej na temat tworzenia kopii obiektu można znaleźć w module 2 tego kursu.

- **IConvertible** – składa się z szeregu metod, przy pomocy których konwertujemy obiekt naszej klasy do wartości jednego z typów wbudowanych: `char`, `int`, `string` itp. oraz struktury `DateTime`. W przypadku gdy nasz typ implementuje interfejs **IConvertible**, obiekty tego typu możemy konwertować przy pomocy metod klasy `Convert`.
- **IFormattable** – posiada metodę `ToString`, która służy do utworzenia reprezentacji stanu obiektu w postaci napisu. Jeżeli klasa implementuje ten interfejs, to metoda tego interfejsu jest wywoływana, a nie wirtualna metoda `Object.ToString`, gdy wypisujemy stan obiektu danej klasy przy pomocy metod `Console.Write` czy `Console.WriteLine` lub tworzymy napis przy pomocy metody `String.Format`.
- **ICollection<T>** – definiuje metody i właściwości, które powinna implementować każda kolekcja. Interfejs ten dziedziczy po **IEnumerable<T>** i **IEnumerable**. Oprócz metody `GetEnumerator` posiada on następujące składowe:
 - `Count` – właściwość zwracająca liczbę elementów zawartych w kolekcji
 - `IsReadOnly` – właściwość zwracająca wartość typu logicznego informującą, czy kolekcja jest tylko do odczytu
 - `Add` – metoda dodająca element do kolekcji
 - `Clear` – metoda usuwająca wszystkie elementy z kolekcji
 - `Contains` – metoda sprawdzająca, czy kolekcja zawiera dany element
 - `CopyTo` – metoda kopiująca elementy kolekcji do tablicy
 - `Remove` – metoda usuwająca pierwsze wystąpienie podanego elementu z kolekcji
- **IList<T>** – reprezentuje kolekcję, której elementy są dostępne za pomocą indeksu. Interfejs ten dziedziczy po interfejsie **ICollection<T>**. Oprócz metod i właściwości interfejsu **ICollection<T>** posiada on następujące składowe:
 - `IndexOf` – metoda zwracająca indeks podanego elementu, jeżeli kolekcja nie zawiera danego elementu zwracana jest wartość -1.
 - `Insert` – metoda wstawiająca element do kolekcji na pozycji wskazanej przez indeks
 - `RemoveAt` – metoda usuwająca element z pozycji wskazanej przez indeks
 - Indeksator – metoda zwracająca element o podanym indeksie

Przykładowe rozwiązanie

Implementacja interfejsów **IEnumerable** i **IEnumerator**

Naszym zadaniem jest zaimplementowanie dla listy jednokierunkowej możliwości sekwencyjnego przetwarzania jej elementów. W tym celu dla listy zaimplementujemy interfejs **IEnumerable** i **IEnumerator**. Na początku zaimplementujemy te interfejsy tak, jak to się implementowało w wersji języka C# 1.0. Następnie pokażemy zalety zastosowania składni iteratorów wprowadzonych w wersji 2.0.

Implementacja interfejsów **IEnumerable** i **IEnumerator** „w starym stylu”

Załóżmy, że mamy zdefiniowaną następującą klasę reprezentującą listę jednokierunkową:

```
public class ListaJednokierunkowa1<T> {
    private class Wezel {
        public T dane;
        public Wezel nastepny;
        public Wezel(T x, Wezel nastepny) {
            this.nastepny = nastepny;
            dane = x;
        }
    }

    Wezel glowa = null;
```

```
        public void DodajDoGlowy(T x) {  
            glowa = new Wezel(x, glowa);  
        }  
    }  
}
```

Wewnątrz klasy zdefiniujemy nową klasę implementującą interfejs `IEnumerator<T>`. Może ona być klasą prywatną, gdyż nie będziemy na zewnątrz używać obiektów tej klasy. Obiekty te będą dostępne tylko za pomocą interfejsu `IEnumerator<T>`. Dzięki temu, że klasa jest zdefiniowana jako wewnętrzna, ma dostęp do prywatnych składowych klasy `ListaJednokierunkowa1`. Klasa powinna mieć pole reprezentujące obiekt listy jednokierunkowej, z którym jest powiązana. Dodatkowo klasa będzie posiadać pole typu `Wezel`, które reprezentuje aktualnie wybrany węzeł listy – stan naszego iteratora. Przykładowa implementacja znajduje się poniżej.

```
public class ListaJednokierunkowa1<T> {  
    ...  
    private class ListaJednokierunkowa1Enumerator: IEnumerator<T> {  
        private Wezel tmp = null;  
        private ListaJednokierunkowa1<T> lista;  
  
        public ListaJednokierunkowa1Enumerator(ListaJednokierunkowa1<T> lista)  
        {  
            this.lista = lista;  
        }  
  
        public T Current {  
            get { return tmp.dane; }  
        }  
  
        public void Dispose() {  
        }  
  
        object System.Collections.IEnumerator.Current {  
            get { return Current; }  
        }  
  
        public bool MoveNext() {  
            if (tmp == null)  
                tmp = lista.glowa;  
            else  
                tmp = tmp.nastepny;  
            return tmp != null;  
        }  
  
        public void Reset() {  
            tmp = null;  
        }  
    }  
}
```

Zwróćmy uwagę, że nie rzucamy wyjątku `NotImplementedException` w metodzie `Dispose`, mimo że nie dostarczamy żadnej implementacji tej metody. Rzucenie tego wyjątku w tej metodzie spowodowałoby, że wykonanie pętli `foreach` z naszą kolekcją kończyłoby się zawsze zgłoszeniem wyjątku.

Następnie należy zaznaczyć, że nasza klasa reprezentująca listę jednokierunkową implementuje interfejs `IEnumerable` i zaimplementować metody tego interfejsu:

```

public class ListaJednokierunkowa1<T> : IEnumerable<T> {
    ...
    public IEnumerator<T> GetEnumerator() {
        return new ListaJednokierunkowa1Enumerator(this);
    }

    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator() {
            return GetEnumerator();
        }
}

```

Implementacja interfejsów IEnumerable i IEnumerator „w nowym stylu”

Założmy że mamy zdefiniowaną klasę reprezentującą listę jednokierunkową w identyczny sposób, jak to pokazano na początku tego rozdziału. Zmieńmy nazwę tylko tej klasy na `ListaJednokierunkowa2`. Zaznaczmy, że nasza klasa implementuje interfejs `IEnumerable`. Od wersji języka C# 2.0 nie musimy pisać klasy implementującej interfejs `IEnumerator`. Klasa ta zostanie automatycznie wygenerowana przez kompilator, jeżeli napotka on słowo `yield`.

```

public class ListaJednokierunkowa2<T> : IEnumerable<T> {
    ...
    public IEnumerator<T> GetEnumerator() {
        Wezel tmp = glowa;
        while (tmp != null) {
            yield return tmp.dane;
            tmp = tmp.nastepny;
        }
    }

    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator() {
            return GetEnumerator();
        }
}

```

Jak widać powyższa implementacja jest dużo krótsza i chyba prostsza.

Można dla jednej klasy kolekcji zaimplementować alternatywne sposoby przemieszczania się po jej elementach. Założmy, że chcemy otrzymywać co drugi element naszej kolekcji. W tym celu musimy utworzyć klasę implementującą interfejs `IEnumerable`, a następnie metodę lub właściwość, która zwróci obiekt tej klasy. Przykładowa implementacja jest umieszczona poniżej:

```

private class ListaJednokierunkowa2<T> : IEnumerable<T> {
    ...
    public class CoDrugiElement : IEnumerable<T> {
        private ListaJednokierunkowa2<T> lista;
        public CoDrugiElement(ListaJednokierunkowa2<T> lista) {
            this.lista = lista;
        }

        public IEnumerator<T> GetEnumerator() {
            Wezel tmp = lista.glowa;
            while (tmp != null) {
                tmp = tmp.nastepny;
                if (tmp != null) {
                    yield return tmp.dane;
                    tmp = tmp.nastepny;
                }
            }
        }
    }
}

```

```
    }

    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}

public IEnumerable<T> CoDrugi {
    get { return new CoDrugiElement(this); }
}
}
```

Użycie interfejsów IEnumerable i IEnumerator

Z interfejsów IEnumerable i IEnumerator możemy skorzystać w sposób bezpośredni, co ilustruje poniższy przykład:

```
ListaJednokierunkowa1<string> zoo1 = new ListaJednokierunkowa1<string>();
zoo1.DodajDoGlowy("Wilk");
zoo1.DodajDoGlowy("Lew");
zoo1.DodajDoGlowy("Tygrys");

IEnumerator<string> ie1 = zoo1.GetEnumerator();
while (ie1.MoveNext()) {
    Console.Write("{0}, ", ie1.Current);
}
```

Możemy je też wykorzystać pośrednio, korzystając z pętli foreach:

```
foreach (string s in zoo1) {
    Console.Write("{0}, ", s);
}
```

Oba sposoby możemy wykorzystać również wtedy, gdy interfejsy zaimplementujemy przy pomocy składni iteratorów:

```
ListaJednokierunkowa2<string> imiona = new ListaJednokierunkowa2<string>();
imiona.DodajDoGlowy("Adam");
imiona.DodajDoGlowy("Karol");
imiona.DodajDoGlowy("Piotr");
imiona.DodajDoGlowy("Anna");
imiona.DodajDoGlowy("Ewa");
imiona.DodajDoGlowy("Dorota");

IEnumerator<string> ie2 = imiona.GetEnumerator();
while (ie2.MoveNext()) {
    Console.Write("{0}, ", ie2.Current);
}
Console.WriteLine();
foreach (string s in imiona) {
    Console.Write("{0}, ", s);
}
```

Jeżeli chcemy otrzymać tylko co drugi element kolekcji, możemy skorzystać z właściwości CoDrugi:

```
foreach (string s in imiona.CoDrugi) {
    Console.Write("{0}, ", s);
}
```

Implementacja interfejsu *IFormattable*

Naszym zadaniem jest dodanie do klasy *Osoba* możliwości przedstawienia jej stanu w postaci napisu na dwa sposoby:

- wersja skrócona, w utworzonym napisie jest uwzględnione tylko imię i nazwisko
- wersja pełna, w utworzonym napisie uwzględniane są wszystkie pola (właściwości)

Implementacja klasy *Osoba*

Założmy, że mamy zdefiniowaną następującą klasę:

```
class Osoba {  
    public string Imie { set; get; }  
    public string Nazwisko { set; get; }  
    public int RokUrodzenia { set; get; }  
}
```

W celu dostarczenia różnych napisów reprezentujących stan obiektu klasy *Osoba* musimy w tej klasie zaimplementować interfejs *IFormattable*. Interfejs ten posiada tylko pojedynczą metodę *ToString*. W zależności od wartości pierwszego argumentu metoda ta będzie zwracać napis zawierający lub nie rok urodzenia danej osoby. W przypadku gdy argument ten będzie miał wartość *null* lub będzie reprezentować napis pusty, stan obiektu będzie reprezentowany tylko przez imię i nazwisko. Natomiast gdy wartość tego argumentu będzie równa „p” lub „P” (pierwsza litera słowa „pełny”), w stanie obiektu będzie również uwzględniony rok urodzenia. W pozostałych przypadkach należy rzucić wyjątek *FormatException*. Metodę *IFormattable.ToString* uczynimy również wirtualną, aby móc ją nadpisać w klasach pochodnych po klasie *Osoba*.

```
class Osoba : IFormattable {  
    ...  
    public virtual string ToString(string format,  
                                   IFormatProvider formatProvider) {  
        if (string.IsNullOrEmpty(format)) {  
            return string.Format("{0} {1}", Imie, Nazwisko);  
        }  
        if (format == "p" || format == "P") {  
            return string.Format("{0} {1} ur. {2}r.",  
                                   Imie, Nazwisko, RokUrodzenia);  
        }  
        throw new FormatException("Niepoprawne formatownie: " + format);  
    }  
}
```

Użycie interfejsu *IFormattable*

Interfejs *IFormattable* jest wykorzystywany przez metody *Write* i *WriteLine* klasy *Console*:

```
Osoba o = new Osoba {  
    Imie = "Jan", Nazwisko = "Kowalski", RokUrodzenia = 1989 };  
  
Console.WriteLine("Pan {0}", o);  
Console.WriteLine("Pan {0:p}", o);
```

Interfejs ten jest również wykorzystywany w przypadku metody *string.Format*:

```
string s1 = string.Format("Pan {0}", o);  
string s2 = string.Format("Pan {0:P}", o);  
Console.WriteLine(s1);  
Console.WriteLine(s2);
```

Oczywiście poniższy kod spowoduje zgłoszenie wyjątku:

```
string s3 = string.Format("Pan {0:f}", o);
```


Gotowy rozwiązany powyższy przykład znajduje się w katalogu **Demo\Modul11**.

Porady praktyczne

- Nie twórz własnych interfejsów, jeżeli już istnieje standardowy interfejs zdefiniowany w bibliotece .NET Framework określający podobną funkcjonalność. Tworząc nowy interfejs nie dość, że musisz wykonać dodatkową pracę, powodujesz, że Twoje klasy są mniej intuicyjne i trudniejsze w użyciu.
- Zanim utworzysz finalizator dla swojej klasy, zastanów czy jest on na pewno konieczny. Dodanie finalizatora powoduje, że garbage collector musi najpierw umieścić obiekt Twojej klasy w kolejce obiektów oczekujących na finalizację. Dopiero po wywołaniu finalizatora pamięć zajmowana przez obiekt może zostać zwolniona. Powoduje to, że do zwolnienia pamięci zajmowanej przez taki obiekt potrzeba co najmniej dwóch cykli działania garbage collector.
- Finalizator możesz tworzyć tylko dla klas, definicja finalizatora dla struktur jest niedozwolona. Możesz jednak w swoich strukturach zaimplementować interfejs `IDisposable`.
- Definiując finalizator nie możesz przysyłać do niego żadnych argumentów. Finalizator nie może być wywołany w sposób jawny. W definicji finalizatora nie możesz stosować żadnych modyfikatorów (`protected`, `public`, `virtual`, `override` itp.).
- Definiując finalizator dla danej klasy, zaimplementuj również dla niej klasy interfejs `IDisposable`, aby dać użytkownikowi możliwość deterministycznego zwalniania zasobów zajmowanych przez obiekty danej klasy.
- Wewnątrz metody `Dispose` nie zapomnij wywołać metody `GC.SuppressFinalize(this)`, w celu poinformowanie garbage collector, że zasoby używane przez obiekt zostały zwolnione i pamięć zajmowana przez ten obiekt może zostać od razu zwolniona – nie trzeba czekać na zakończenie procesu finalizacji.
- Pamiętaj, że zmienna w pętli `foreach` reprezentująca aktualnie przetwarzany element kolekcji jest tylko do odczytu. W przypadku gdy kolekcja zawiera elementy typu referencyjnego, możesz jednak modyfikować pola obiektu, do którego ta zmienna zawiera odwołanie.
- Pętle `foreach` możesz stosować tylko do kolekcji, dla których ich typ implementuje interfejs `IEnumerable` lub zawiera publiczną metodę `GetEnumerator`, której typ zwracany określony jest przez interfejs `IEnumerator`.
- Implementuj interfejsy `IEnumerable` i `IEnumerator` zawsze w wersji generycznej. Korzystając z właściwości `Current` nie będziesz musiał dokonywać konwersji z typu `object` na typ podstawowy kolekcji, co niewątpliwie poprawi wydajność Twoich programów.
- Implementując również interfejs `IComparable` korzystaj z wersji generycznej. Wewnątrz metody `CompareTo` nie będziesz musiał sprawdzać, czy przesłany obiekt do metody jest odpowiedniego typu, co wpłynie na efektywność tej metody.
- W przypadku gdy przeciążasz operatory porównania, również w danej klasie lub strukturze zaimplementuj interfejs `IComparable`. Implementacja tego interfejsu jest sposobem bardziej elastycznym. Implementując ten interfejs będziesz mógł sortować predefiniowane generyczne kolekcje oraz tablice elementów typu implementującego ten interfejs.
- Alternatywny sposób porównania obiektów danej klasy możesz zapewnić przez implementację interfejsu `IComparer`. Interfejs ten implementuj w klasie zdefiniowanej wewnątrz klasy, której obiekty chcesz porównywać.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- wiesz, co to jest finalizator

- wiesz do czego służy i jak zaimplementować interfejs `IDisposable`
- wiesz, co to jest blok instrukcji `using` i potrafisz z niego skorzystać
- znasz pojęcie wzorca projektowego iterator
- wiesz do czego służą i jak zaimplementować interfejsy `IEnumerable` i `IEnumerator`
- potrafisz użyć konstrukcji iteratorów do implementacji interfejsu `IEnumerable`
- wiesz do czego służą i jak zaimplementować interfejsy `IComparable` i `IComparer`
- znasz przeznaczenie następujących interfejsów: `IList<T>`, `ICollection<T>`, `IFormattable`, `IConvertible`, `ICloneable`, `IEqualityComparer<T>` i `IEquatable<T>`

Dodatkowe źródła informacji

1. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka skierowana do programistów chcących nauczyć się programować w języku C#.

2. Andrew Troelsen, *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition*, Apress, 2007

Książka przeznaczona dla bardziej zaawansowanych programistów. Czwarte wydanie opisuje język C# 3.0 i platformę .NET 3.5.

3. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad jak powinniśmy pisać swoje programy.

4. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Wzorce projektowe Wydanie II*, WNT, 2008

Książka, która wprowadziła pojęcie wzorców projektowych do informatyki. Lektura obowiązkowa dla każdego, kto chce poznać temat wzorców projektowych.

5. Steven John Metsker, *C#. Wzorce projektowe*, Helion, 2005

Książka jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia jak wykorzystać cechy języka C# do tworzenia poprawnego kodu poprzez zastosowanie wzorców.

6. Judith Bishop, *C# 3.0 Design Patterns*, O'Reilly Media, Inc. 2008

Książka, podobnie jak poprzednia, jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia również nowe cechy języka C#.

7. Codeguru, <http://www.codeguru.pl>

Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.

8. C Sharp Tutorial, http://www.meshplex.org/wiki/C_Sharp_Tutorial

Internetowy kurs języka C#.

9. C# Corner, <http://www.csharpcorner.com>

Portal poświęcony programowaniu w języku C#.

10. Kurs C#, cz. I, http://www.centrumxp.pl/dotNet/20,1,kategoria,Kurs_C_cz_I.aspx


Przystępny kurs języka C# w języku polskim.

Laboratorium podstawowe

Problem 1 (czas realizacji 30 min)

Masz znajomego, który prowadzi komis samochodowy. Jego klienci przy wyborze samochodu biorą pod uwagę różne kryteria. Jednych interesuje rok produkcji, innych cena lub model samochodu. Zostałeś poproszony o utworzenie programu, który wyświetlałby dostępne samochody i sortował je według następujących kryteriów:

- nazwa marki
- nazwa modelu
- rok produkcji
- cena

Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition	<ul style="list-style-type: none"> • Otwórz Visual C# 2008 Express Edition. • Z menu wybierz File -> New Project. • Z listy Visual Studio installed templates wybierz Class Library. • W polu Name wpisz Samochody. • Kliknij OK. • Z menu wybierz File -> Save Samochody. • W polu Location wybierz folder w którym będzie zapisany projekt. • Zaznacz pole wyboru Create directory for solution. • W polu Solution Name wpisz Modul11. • Naciśnij przycisk Save.
2. Utwórz klasę Samochod, która będzie implementować interfejs IComparable	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na Samochod.cs. W okienku dialogowym naciśnij przycisk Tak. • Zaznacz, że klasa Samochod implementuje interfejs IComparable: <pre>public class Samochod : IComparable<Samochod> { }</pre>
3. Do klasy Samochod dodaj konieczne właściwości	<ul style="list-style-type: none"> • Do klasy Samochod dodaj właściwości reprezentujące cechy takie jak marka, model, cena i rok produkcji: <pre>public class Samochod : IComparable<Samochod> { public string Marka { set; get; } public string Model { set; get; } public int RokProdukcji { set; get; } public decimal Cena { set; get; } }</pre>
4. W klasie Samochod nadpisz metodę ToString	<ul style="list-style-type: none"> • W klasie Samochod nadpisz metodę wirtualną ToString: <pre>public override string ToString() { return string.Format("Samochód {0,-10} {1,-10} cena: {2,-10} " + "rok produkcji {3,-10}", Marka, Model, Cena, RokProdukcji); }</pre>
5. Do klasy Samochod dodaj implementację interfejsu IComparable	<ul style="list-style-type: none"> • W klasie Samochod dodaj metodę CompareTo, wymaganą przez interfejs IComparable: <pre>public int CompareTo(Samochod other) { return Marka.CompareTo(other.Marka); }</pre> <p> W celu implementacji metod interfejsu w danej klasie, umieść punkt wstawiania (kursor klawiatury) w nazwie żądanego interfejsu na liście</p>

	<p>dziedziczenia tej klasy. Pod nazwą interfejsu powinna pojawić się wtedy cienka kreseczka (podobna do opcji autokorekty). Przy pomocy wskaźnika myszy możesz rozwinąć tą kresczkę do menu kontekstowego. Z menu możesz wybrać Implement interface <nazwa interfejsu>, gdy chcemy zaimplementować metody interfejsu w zwykły sposób lub Explicitly implement interface <nazwa interfejsu>, gdy chcemy zaimplementować metody interfejsu w sposób jawny. Po wybraniu jednej z pozycji menu kontekstowego, Visual Studio wygeneruje szkielety metod określonych przez dany interfejs.</p>
6. W klasie Samochod zdefiniuj klasę PoModelu	<ul style="list-style-type: none"> W klasie Samochod zdefiniuj publiczną klasę PoModelu. Klasa ta będzie implementować interfejs IComparer. W metodzie Compare jako kryterium porównawcze wybierz właściwość Model: <pre>public class PoModelu : IComparer<Samochod> { public int Compare(Samochod x, Samochod y) { return string.Compare(x.Model, y.Model, true); } }</pre>
7. W klasie Samochod zdefiniuj klasę PoCenie	<ul style="list-style-type: none"> W klasie Samochod zdefiniuj publiczną klasę PoCenie. Klasa ta będzie implementować interfejs IComparer. W metodzie Compare jako kryterium porównawcze wybierz właściwość Cena: <pre>public class PoCenie : IComparer<Samochod> { public int Compare(Samochod x, Samochod y) { return x.Cena.CompareTo(y.Cena); }; }</pre>
8. W klasie Samochod zdefiniuj klasę PoRokuProdukcji	<ul style="list-style-type: none"> W klasie Samochod zdefiniuj publiczną klasę PoRokuProdukcji. Klasa ta będzie implementować interfejs IComparer. W metodzie Compare jako kryterium porównawcze wybierz właściwość RokProdukcji: <pre>public class PoRokuProdukcji : IComparer<Samochod> { public int Compare(Samochod x, Samochod y) { return x.RokProdukcji.CompareTo(y.RokProdukcji); } }</pre>
9. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestSamochodu. Kliknij OK.
10. Zaznacz projekt TestSamochodu jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestSamochodu, a następnie z menu kontekstowego wybierz Set as StartUp Project.
11. Do projektu TestSamochodu dodaj odwołanie do biblioteki Samochody	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projektu TestSamochodu, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt Samochody, a następnie kliknij przycisk OK.
12. Zaimplementuj program, który umożliwia	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Na górze pliku Program.cs zaimportuj przestrzeń nazw Samochody:

dodawanie samochodów oraz wyświetlanie ich w określonym porządku

```
using Samochody;
```

- Do klasy **Program** dodaj statyczną metodę, która wypisuje wszystkie samochody zawarte w danym komisie:

```
public static void WypiszSamochody(List<Samochod> komis) {
    Console.Clear();
    for (int i = 1; i <= komis.Count; i++) {
        Console.WriteLine("{0,3}. {1}", i, komis[i-1]);
        if (i % Console.WindowHeight == 0)
            Console.ReadKey(false);
    }
    Console.ReadKey();
}
```

- Do klasy **Program** dodaj statyczną metodę, która wyświetla dostępne opcje programu:

```
static char Menu() {
    Console.Clear();
    Console.WriteLine("A - Dodaj samochód");
    Console.WriteLine("B - Wypisz według nazwy marki");
    Console.WriteLine("C - Wypisz według roku produkcji");
    Console.WriteLine("D - Wypisz według nazwy modelu");
    Console.WriteLine("E - Wypisz według ceny");
    Console.WriteLine("K - Koniec");
    return Console.ReadKey().KeyChar;
}
```

- Do klasy **Program** dodaj statyczną metodę, która tworzy nowy obiekt klasy **Samochod** pobierając odpowiednie informacje od użytkownika:

```
private static Samochod DodajSamochod() {
    Console.WriteLine("Podaj markę samochodu: ");
    string marka = Console.ReadLine();
    Console.WriteLine("Podaj model samochodu: ");
    string model = Console.ReadLine();
    decimal cena;
    do {
        Console.WriteLine("Podaj cenę samochodu: ");
    }
    while (!decimal.TryParse(Console.ReadLine(), out cena));
    int rokProdukcji;
    do {
        Console.WriteLine("Podaj rok produkcji samochodu: ");
    }
    while (!int.TryParse(Console.ReadLine(), out rokProdukcji));

    return new Samochod() {
        Cena = cena,
        RokProdukcji = rokProdukcji,
        Model = model,
        Marka = marka
    };
}
```

- Zaimplementuj metodę **Main**:

```
static void Main(string[] args) {
    List<Samochod> komis = new List<Samochod>(100) {
        new Samochod() {
            Marka="Audi", Model = "A3", RokProdukcji=2007, Cena=100000 },
        new Samochod() {
            Marka="Opel", Model = "Vectra", RokProdukcji=2000,
            Cena=10000 },
        new Samochod() {
            Marka="Syrena", Model = "105L", RokProdukcji=1978, Cena=100 },
    }
```

	<pre> new Samochod() { Marka="Fiat", Model = "125", RokProdukcji=1980, Cena=105 } }; char c; do { c = Menu(); switch (c) { case 'a': case 'A': komis.Add(DodajSamochod()); break; case 'b': case 'B': komis.Sort(); WypiszSamochody(komis); break; case 'c': case 'C': komis.Sort(new Samochod.PoRokuProdukcji()); WypiszSamochody(komis); break; case 'd': case 'D': komis.Sort(new Samochod.PoModelu()); WypiszSamochody(komis); break; case 'e': case 'E': komis.Sort(new Samochod.PoCenie()); WypiszSamochody(komis); break; } } while (!(c == 'k' c == 'K')); } </pre>
13. Skompiluj i uruchom program	<ul style="list-style-type: none"> • Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. • W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Problem 2 (czas realizacji 15 min)

W pewnym programie tworzonym przez firmę, w której pracujesz, zostało postawione wymaganie, aby wszystkie operacje wykonywane przez użytkownika były logowane. Postanowiono więc utworzyć klasę, przy pomocy której konieczne informacje będą wyświetlane w konsoli oraz pobierane od użytkownika i jednocześnie zapisywane do pliku. Ze względu na ważność zadania, Ty – jako jeden z najlepszych programistów w firmie – zostałeś wybrany do stworzenia tej klasy.

Zadanie	Tok postępowania
1. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. • Z listy Visual Studio installed templates wybierz Class Library. • W polu Name wpisz RozszerzeniaKonsoli. • Kliknij OK.
2. Utwórz klasę RozszerzonaKonsola	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na RozszerzonaKonsola.cs. W okienku dialogowym naciśnij przycisk Tak.
3. Zaimplementuj klasę	<ul style="list-style-type: none"> • W pliku RozszerzonaKonsola.cs zaimportuj przestrzeń nazw System.IO: using System.IO;

RozszerzonaKonsola	<ul style="list-style-type: none"> Do klasy RozszerzonaKonsola dodaj prywatne pole reprezentujące strumień, do którego będziemy zapisywać aktywność użytkownika: <pre>private StreamWriter sw;</pre> Do klasy RozszerzonaKonsola dodaj publiczny konstruktor, do którego będziemy przysyłać nazwę pliku wraz z ścieżką, do którego będziemy zapisywać aktywność użytkownika: <pre>public RozszerzonaKonsola(string scizka) { sw = new StreamWriter(scizka); }</pre> Do klasy RozszerzonaKonsola dodaj publiczne metody, przy pomocy których będziemy mogli nawiązać konwersacje z użytkownikiem z jednoczesnym zapisem jej w pliku: <pre>public void WriteLine(string s) { Console.WriteLine(s); sw.WriteLine(s); } public void WriteLine(string s, params object[] args) { Console.WriteLine(s, args); sw.WriteLine(s, args); } public void Write(string s) { Console.Write(s); sw.Write(s); } public void Write(string s, params object[] args) { Console.Write(s, args); sw.Write(s, args); } public string ReadLine() { string s = Console.ReadLine(); sw.WriteLine("###{0}###", s); return s; } public int Read() { int n = Console.Read(); sw.WriteLine("\$\$\${0}\$\$\$", n); return n; }</pre>
4. W klasie RozszerzonaKonsola zaimplementuj interfejs IDisposable	<ul style="list-style-type: none"> Zaznacz, że klasa RozszerzonaKonsola implementuje interfejs IDisposable: <pre>public class RozszerzonaKonsola: IDisposable</pre> Do klasy RozszerzonaKonsola dodaj pole chronione typu logicznego, które będzie informować, czy strumień już został zamknięty: <pre>protected bool disposed = false;</pre> Do klasy RozszerzonaKonsola dodaj wirtualną metodę chronioną, która będzie zamykać strumień używany przez obiekt tej klasy: <pre>protected virtual void Dispose(bool disposing) { if (disposed) return; if (disposing) {</pre>

	<pre> sw.Dispose(); } disposed = true; } </pre> <ul style="list-style-type: none"> Do klasy RozszerzonaKonsola dodaj metodę Dispose implementującą interfejs IDisposable: <pre> public void Dispose() { Dispose(true); GC.SuppressFinalize(this); } </pre> Do klasy RozszerzonaKonsola dodaj finalizator: <pre> ~RozszerzonaKonsola() { Dispose(false); } </pre> Do klasy RozszerzonaKonsola dodaj publiczną właściwość tylko do odczytu zwracającą informację czy strumień został już zamknięty: <pre> public bool IsDisposed { get { return disposed; } } </pre>
5. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestKonsoliRozszerzonej. Kliknij OK.
6. Zaznacz projekt TestKonsoliRozszerzonej jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestKonsoliRozszerzonej, a następnie z menu kontekstowego wybierz Set as StartUp Project.
7. Do projektu TestKonsoliRozszerzonej dodaj odwołanie do biblioteki Sortowanie	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projektu TestKonsoliRozszerzonej, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt RozszerzeniaKonsoli, a następnie kliknij przycisk OK.
8. Zaimplementuj metodę Main	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Na górze pliku Program.cs zaimportuj przestrzeń nazw RozszerzeniaKonsoli: <pre> using RozszerzeniaKonsoli; </pre> Przetestuj działanie klasy RozszerzonaKonsola: <pre> static void Main(string[] args) { using (RozszerzonaKonsola konsola = new RozszerzonaKonsola("zapis.txt")) { konsola.WriteLine("Witamy w programie"); konsola.Write("Podaj imię: "); string imie = konsola.ReadLine(); konsola.Write("Podaj nazwisko: "); string nazwisko = konsola.ReadLine(); konsola.WriteLine("Czesc {0} {1}", imie, nazwisko); } Console.ReadKey(); } </pre>

9. Skompiluj i uruchom program	<ul style="list-style-type: none">• Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program.• W celu uruchomienia programu, z menu Debug wybierz Start Debugging.
--------------------------------	---

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 90 min)

W firmie stworzono klasę reprezentującą drzewo binarne. Klasa ta jednak nie implementuje żadnych standardowych interfejsów, nie można z niej korzystać w standardowy sposób, tak jak z innych predefiniowanych kolekcji. Musisz dostosować tę klasę do standardów obowiązujących kolekcje w .NET Framework. Pliki startowe do tego zadania znajdują się w katalogu **Kurs\Lab\Start\Modul11\DrzewoBinarne.cs**, gdzie **Kurs** jest katalogiem, w którym zainstalowano pliki kursu.

Drzewo binarne jest omówione w kursie „Wprowadzenie do programowania”.

Twoim zadaniem jest zaimplementowanie dla klasy `DrzewoBinarne<T>` następujących interfejsów:

- `IEnumerable<T>` oraz `IEnumerator<T>` – zwróć elementy od najmniejszego do największego.
- `ICollection<T>`
- `IList<T>` – w metodzie `Insert` zgłoś wyjątek `NotSupportedException`.
- `ICloneable` – utwórz nowe drzewo, które będzie przechowywać te same obiekty, co drzewo oryginalne, ale nie będzie miał żadnego wspólnego węzła z oryginalnym drzewem.
- `IComparable` – porównuj kolejne elementy w drzewach aż do napotkania różnych elementów lub wyczerpania się elementów w jednym z drzew. Kolejność jest wyznaczona przez interfejs `IEnumerator`, czyli od najmniejszego do największego.

Napisz również program, który pokazuje, jak można skorzystać ze zmodyfikowanej klasy `DrzewoBinarne<T>`.

Zadanie 2 (czas realizacji 60 min)

W firmie stworzono strukturę reprezentującą liczby zespolone. Typ ten jednak nie implementuje żadnych standardowych interfejsów, przez co nie można z niego korzystać w standardowy sposób, tak jak z innych typów wbudowanych. Musisz dostosować ten typ do standardów obowiązujących w .NET Framework. Pliki startowe do tego zadania znajdują się w katalogu **Kurs\Lab\Start\Modul11\LiczbaZespolona.cs**, gdzie **Kurs** jest katalogiem, w którym zainstalowano pliki kursu.

Twoim zadaniem jest zaimplementowanie w strukturze `Zespolona` następujących interfejsów:

- `IConvertible`
- `IFormattable` – uwzględnij następujące możliwości przedstawienia liczby zespolonej:
 - tylko część rzeczywista
 - tylko część urojona
 - postać algebraiczna
 - postać trygonometryczna
 - postać wykładnicza
- `IEquatable<T>`

Napisz również program, który pokazuje, jak można skorzystać ze zmodyfikowanej struktury `Zespolona`.

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 12

Wersja 2

Delegacje i zdarzenia

Spis treści

Delegacje i zdarzenia.....	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	8
Porady praktyczne	10
Uwagi dla studenta	11
Dodatkowe źródła informacji	11
Laboratorium podstawowe	13
Problem 1 (czas realizacji 15 min)	13
Problem 2 (czas realizacji 30 min)	15
Laboratorium rozszerzone	20
Zadanie 1 (czas realizacji 90 min)	20

Informacje o module

Opis modułu

W tym module zapoznasz się z pojęciami delegacji i zdarzenia. Dowiesz się, co to są delegacje i jak się je definiuje, zarówno w standardowy sposób, jak i przy pomocy metod anonimowych i wyrażeń lambda. Nauczysz się również definiować własne zdarzenia.

Cel modułu

Celem modułu jest przedstawienie, co to są delegacje i zdarzenia oraz jak je definiujemy i używamy w języku C#.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- wiedział, co to jest delegacja
- wiedział, co to jest zdarzenie
- potrafił tworzyć delegacje zarówno przy pomocy metod anonimowych, jak i wyrażeń lambda
- umiał definiować własne zdarzenia

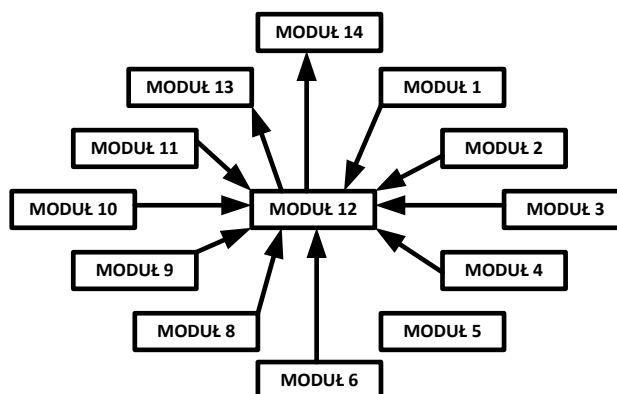
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinienś:

- rozumieć pojęcie klasy
- rozumieć pojęcie metody
- potrafić definiować własne metody
- wiedzieć, co to są metody statyczne

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w modułach „Pojęcie klasy”, „Konstruktor”, „Właściwości i indeksatory”, „Składowe statyczne”, „Dziedziczenie”, „Polimorfizm – funkcje wirtualne”, „Klasy abstrakcyjne i interfejsy”, „Typy ogólne” oraz „Implementacja przykładowych interfejsów”.



Rys. 18 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Otrzymałeś ostatnio zadania napisania pewnej klasy. Klasa ma sterować pracą różnych urządzeń. Użytkownik ma podać godzinę, o której powinno zostać uruchomione urządzenie i obiekt Twojej klasy ma wywołać o danej godzinie odpowiednią funkcję, która uruchomi dane urządzenie. Zapytałeś, jakie urządzenie masz uruchomić i gdzie są do niego sterowniki. Szef popatrzył na Ciebie dziwnie i niezbyt grzecznie powiedział, że nie powinno Ciebie to obchodzić. Po chwili dodał, że Twoja klasa będzie używana w różnych projektach, do uruchamiania różnych rzeczy i Twoim jedynym zadaniem jest zapewnienie, aby funkcje dostarczone przez programistów piszących poszczególne projekty były wywoływane o odpowiednim czasie. Miałeś już zapytać, jak wywołać funkcję, która jeszcze nie istnieje, ale ugryzłeś się w język. Masz przecież zamiar porozmawiać o podwyżce, a przyznanie się do braku wiedzy nie będzie dobrym argumentem przy rozmowie z szefem. Postanowiłeś najpierw się dowiedzieć. Masz szczęście. Moduł ten pokaże Ci, jak można w metodach swoich klas wywoływać funkcje, które w momencie tworzenia Twojej klasy nie istnieją.

Podstawy teoretyczne

Delegacje

Delegacje lub inaczej *delegaty* są kolejnym typem referencyjnym w .NET Framework. Obiekt, którego typ określony jest przez delegację, może reprezentować metodę lub metody, innymi słowy obiekt ten zawiera kolekcję metod. Przy pomocy zmiennej zawierającej referencję do takiego obiektu możemy wywołać metodę(y), którą(e) dany obiekt reprezentuje. Zmienna w czasie swojego życia może zawierać referencję do różnych obiektów, więc przy pomocy tej samej zmiennej możemy wywołać różne metody lub różne grupy metod. Co więcej, dzięki temu że metoda może być reprezentowana przez obiekt, metoda niejako może być również parametrem metody. Zaczniemy jednak od początku, delegację, czyli nowy typ, definiujemy przy pomocy słowa kluczowego *delegate*:

```
[modyfikator_dostępu] delegate typ_zwracany  
                                NazwaDelegacji(lista_argumentów);
```

Definiując delegację opisujemy, jakie metody mogą być reprezentowane przez obiekty, których typ określony jest przez daną delegację. Metoda, która może być reprezentowana przez obiekt, którego typ jest określony przez pewną delegację, musi być zgodna co do argumentów (typ, ilość) jak i typu wartości zwracanej. Założmy, że mamy następującą definicję:

```
delegate void MojaDelegacja(string s, int i);
```

Obiekty typu *MojaDelegacja* mogą reprezentować dowolną metodę nie zwracającą żadnej wartości (tj. posiadającą typ wartości zwracanej *void*) i przyjmującą dwa argumenty, pierwszy typu *string*, a drugi typu *int*. W celu pokazania działania delegacji założmy, że mamy zdefiniowaną następującą klasę:

```
class Test {  
    public string Nazwa { set; get; }  
  
    public static void F(string napis, int n) {  
        Console.WriteLine("Metoda statyczna z parametrami: {0} oraz {1}",  
                           napis, n);  
    }  
  
    public void G(string napis, int n) {  
        Console.WriteLine("Metoda zwykła z parametrami: {0} oraz {1}" +  
                           ", na rzecz obiektu {2}", napis, n, Nazwa);  
    }  
}
```

```
    }  
}
```

Zauważmy, że zarówno metoda statyczna F, jak i metoda G są zgodne z delegacją MojaDelegacja. Mają ten sam zestaw argumentów, jak i ten sam typ wartości zwracanej.

Utwórzmy teraz obiekt typu MojaDelegacja reprezentujący metodę statyczną Test.F i przy pomocy tego obiektu wywołajmy metodę z nim powiązaną:

```
MojaDelegacja d1 = new MojaDelegacja(Test.F);  
d1("wywołanie", 1);
```

W wyniku powyższego kodu na ekranie zostanie wyświetlony następujący tekst:

```
Metoda statyczna z parametrami wywołanie oraz 1
```

W celu powiązania obiektu jakiejś delegacji ze zwykłą metodą klasy, należy najpierw utworzyć obiekt, na rzecz którego dana metoda będzie wywołana, co ilustruje poniższy kod:

```
Test t1 = new Test() { Nazwa = "Obiekt1" };  
MojaDelegacja d2 = new MojaDelegacja(t1.G);  
d2("wywołanie", 2);
```

Spowoduje on, że na ekranie pojawi się następujący napis:

```
Metoda zwykła z parametrami wywołanie oraz 2, na rzecz obiektu Obiekt1
```

Od wersji 2.0 biblioteki .NET Framework do tworzenia obiektu delegacji możemy stosować skróconą notację:

```
d1 = Test.F;  
d2 = t1.G;
```

W celu utworzenia obiektu delegacji reprezentującego kilka metod możemy zastosować operator +=:

```
MojaDelegacja d3 = new MojaDelegacja(Test.F);  
d3 += new MojaDelegacja(t1.G);  
d3("wywołanie", 3);
```

W wyniku tego na ekranie zostanie wyświetlony następujący tekst:

```
Metoda statyczna z parametrami wywołanie oraz 3  
Metoda zwykła z parametrami wywołanie oraz 3, na rzecz obiektu Obiekt1
```

Mając delegację reprezentującą kilka metod możemy utworzyć nowy obiekt delegacji, który powstanie niejako przez usunięcie jednej z metod. Używamy do tego operatora -=:

```
d3 -= new MojaDelegacja(Test.F);  
d3("wywołanie", 4);
```

W wyniku powyższego kodu na ekranie zostanie wyświetlony następujący tekst:

```
Metoda zwykła z parametrami wywołanie oraz 4, na rzecz obiektu Obiekt1
```

Metody anonimowe

W wersji 2.0 języka C# wprowadzono nową konstrukcję językową nazwaną *metodą anonimową*. Konstrukcja ta umożliwia utworzenie obiektu delegacji bez konieczności wcześniejszego definiowania metody. Kod, który ma reprezentować delegacja, podajemy w miejscu utworzenia obiektu danej delegacji. Do utworzenia metody anonimowej używamy słowa *delegate*:

```
MojaDelegacja d4 = delegate(string s1, int x) {  
    Console.WriteLine("Metoda anonimowa z parametrami {0} oraz {1}", s1, x);  
};  
d4("wywołanie", 5);
```

W wyniku powyższego kodu na ekranie zostanie wyświetlony następujący tekst:

Metoda anonimowa z parametrami wywołanie oraz 5

Kiedy kompilator wykryje definicję metody anonimowej, automatycznie utworzy nową klasę i skonstruuje wewnątrz niej metodę, która będzie zawierała kod metody anonimowej.

Wyrażenia lambda

W wersji 3.0 języka C# wprowadzono jeszcze jedną konstrukcję językową tzw. wyrażenia lambda. Konstrukcja ta pozwala w łatwy i przejrzysty sposób „na szybko” utworzyć nowy obiekt delegacji i zastępuje niejako metody anonimowe. Wyrażenia lambda wywodzą się z funkcyjnych języków programowania. W językach funkcyjnych definicja funkcji jest identyczna z tą, jaką znamy z matematyki, czyli funkcja jest to przyporządkowanie pewnym parametrom w sposób jednoznaczny określonej wartości. Wyrażenie lambda składa się z listy argumentów, po których następuje operator lambda, a następnie instrukcje, które tworzą naszą funkcję. W języku C# operator lambda stanowi znak równości w połączeniu ze znakiem większości =>:

```
(lista_argumentów) => {instrukcje_definiujące_funkcję};
```

W przypadku gdy funkcja, którą definiujemy przy pomocy wyrażenia lambda przyjmuje tylko jeden argument, nawiasy okrągłe ograniczające argumenty nie są konieczne. Podobnie gdy funkcja składa się tylko z jednej instrukcji, nie musimy stosować nawiasów klamrowych. W celu demonstracji wyrażen lambda rozważmy kilka przykładów. Załóżmy, że mamy zdefiniowane następujące delegacje:

```
delegate double Delgacja1();  
delegate int Delgacja2(int x, int y);  
delegate double Delgacja3(double x);
```

Wykorzystując wyrażenia lambda możemy utworzyć obiekty, których typ jest określony przez poszczególne delegacje, w następujący sposób:

- Funkcja bezparametrowa zawierająca kilka instrukcji kodu:

```
Delgacja1 wyr1 = () => {  
    Console.WriteLine("Funkcja bezparametrowa");  
    return 2.0;  
};  
Console.WriteLine(wyr1());
```

- Powyższy kod spowoduje pojawienie się na ekranie:

```
Funkcja bezparametrowa  
2
```

- Funkcja z kilkoma argumentami. Zwróćmy uwagę, że jeżeli funkcja ma zwracać jakąś wartość i zawiera tylko pojedynczą instrukcję, której nie umieścimy w nawiasach klamrowych, nie stosujemy słowa return. Wartość zwróconą przez funkcję będzie stanowił wartość wyrażenia stanowiącego tą pojedynczą instrukcję.

```
Delgacja2 wyr2 = (a, b) => a + b;  
Console.WriteLine("3 + 4 = {0}", wyr2(3, 4));
```

- Powyższy kod spowoduje pojawienie się na ekranie:

```
3 + 4 = 7
```

- Funkcja z jednym argumentem.

```
Delgacja3 wyr3 = a => a * a;  
Console.WriteLine("Kwadrat liczby 3 wynosi {0}", wyr3(3));
```

- Powyższy kod spowoduje pojawienie się na ekranie:

Kwadrat liczby 3 wynosi 9

Wywołanie asynchroniczne delegacji

Wywołanie asynchroniczne metody, w odróżnieniu od wywołania synchronicznego, umożliwia równoległe wykonanie kodu. W przypadku wywołania synchronicznego wykonanie kodu występującego po wywołaniu metody, następuje dopiero po zakończeniu metody. Natomiast przy wywołaniu asynchronicznym, jedna metoda może wywołać inną metodę, a następnie kontynuować pracę bez oczekiwania na zakończenia działania wywoływanej metody. W wywołaniu asynchronicznym kod metody jest wykonywany w oddzielnym wątku. Jednym ze sposobów wywołania asynchronicznego metody są delegacje asynchroniczne. Do wywołania asynchronicznego używana jest metoda `BeginInvoke`. Do metody tej przekazujemy następujące argumenty:

- Argumenty żądane przez wywoływaną metodę.
- Argument reprezentujący metodę, która będzie wywołana po zakończeniu działania metody wywołanej asynchronicznie. Typ tego argumentu jest określony przez delegację `AsyncCallback`. Parametr ten reprezentuje tzw. *metodę zwrotną* (ang. *callback method*).
- Argument typu `System.Object`, który zawiera wartość przekazywaną do delegacji wywoływanej po zakończeniu wykonywania metody wywołanej asynchronicznie.

W przypadku gdy wywołanie zwrotne nie jest potrzebne, wartości dwóch ostatnich parametrów należy ustawić na `null`. Musimy również pamiętać, że obiekt delegacji, na rzecz którego wywołujemy metodę `BeginInvoke`, może reprezentować tylko pojedynczą metodę. Przykład wywołania asynchronicznego jest umieszczony poniżej:

```
delegate void MojaDelagacja(string s, int i);

static void Test(string s, int n) {
    for (int i = 0; i < n; i++)
        Console.WriteLine(s);
}

static void Main(string[] args) {
    MojaDelagacja d = new MojaDelagacja(Test);
    d.BeginInvoke("Watek 1", 30, null, null);
    d.BeginInvoke("Watek 2", 20, null, null);
    d.BeginInvoke("Watek 3", 10, null, null);
}
```

Należy zwrócić uwagę, że samo zagadnienie programowania asynchronicznego, jest zagadnieniem wykraczającym poza ramy tego kursu.

Zdarzenia

Tworząc własne klasy chcemy wysłać sygnał o zaistnieniu pewnej sytuacji. Programista korzystający z naszej klasy powinien napisać własny kod obsługujący tą sytuację. Częstym przykładem podawanym do opisu takiej sytuacji jest przycisk na formularzu. Naciskając przycisk powodujemy, że uruchamiana jest metoda symulująca wciśnięcie przycisku. Następnie wykonywany jest kod związany z funkcjonalnością danej aplikacji ukryty pod przyciskiem. Kod ten nie jest napisany przez twórcę przycisku, tylko przez autora aplikacji lub formularza. Funkcja rysująca wciśnięty przycisk uruchamia kod dostarczony przez twórcę formularza. Oczywiście, możemy zastosować do tego delegację. Tworząc przycisk, dodajemy do niego publiczne pole, którego typ jest określony przez delegację. Funkcja rysująca wciśnięty przycisk wywołuje również delegację, a co za tym idzie, funkcje, które ona reprezentuje. Wszystko wydaje się odbywać prawidłowo. Jest jednak jeden problem. Skoro pole jest publiczne, to wywołać delegację można również spoza danej klasy, czyli funkcje podłączone pod tę delegację mogłyby być wywołane niezgodnie z zamierzeniem programisty. W celu uniemożliwienia wywołania delegacji spoza klasy wprowadzono pojęcie *zdarzenia*. Zdarzenie definiujemy przy pomocy słowa `event` w następujący sposób:


```
class nazwa_klasy {
    ...
    public event nazwa_delegacji nazwa_zdarzenia;
}
```

Definicja zdarzenia powoduje, że w klasie tworzone jest prywatne pole, którego typ jest określony przez delegację oraz dwie metody umożliwiające sterowanie tym polem. Metody te odpowiednio umożliwiają podłączenie funkcji pod zdarzenie oraz odłączenie funkcji od zdarzenia. Wywołujemy je odpowiednio przy pomocy operatorów `+=` i `-=` i mają taki sam modyfikator dostępu, jaki został użyty przy definicji zdarzenia. W celu wywołania metody lub metod podłączonych pod zdarzenie, wywołujemy zdarzenie przy pomocy jego nazwy, możemy jednak to zrobić tylko z metod klasy, gdzie zdarzenie zostało zdefiniowane, ponieważ pole jest prywatne. Przed wywołaniem zdarzenia należy sprawdzić, czy są podłączone do niego jakieś metody, co pokazuje poniższy przykład:

```
delegate void MojaDelegacja(string s, int i);
class NazwaKlasy {
    ...
    public event MojaDelegacja NazwaZdarzenia;
    public void Metoda() {
        if(NazwaZdarzenia != null)
            NazwaZdarzenia("wywołanie zdarzenia", 2);
    }
}
```

W celu podłączenia się pod zdarzenie należy wykonać następujący kod:

```
NazwaKlasy x = new NazwaKlasy();
x.NazwaZdarzenia += new MojaDelegacja(NazwaMetody);
```

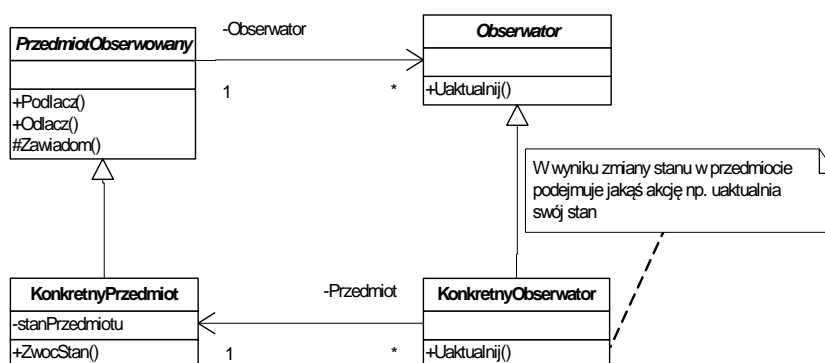
Oczywiście metoda `NazwaMetody` musi być zgodna z delegacją użytą do definicji zdarzenia. Chcąc odłączyć metodę od zdarzenia wykonujemy następujący kod:

```
NazwaKlasy x = new NazwaKlasy();
x.NazwaZdarzenia -= new MojaDelegacja(NazwaMetody);
```

Podobnie jak w poprzednim przykładzie, metoda `NazwaMetody` musi być zgodna z delegacją użytą do definicji zdarzenia.

Wzorzec projektowy Obserwator

Przy pomocy delegacji i zdarzeń w języku C# w łatwy sposób możemy zaimplementować wzorzec projektowy Obserwator (ang. *Observer*). W schematyczny sposób wzorzec ten przedstawiony jest na rys. 19.



Rys. 19 Wzorzec projektowy Obserwator

Wzorzec Obserwator definiuje pomiędzy obiektami relację jeden-do-wielu w taki sposób, że kiedy wybrany obiekt zmienia swój stan, to wszystkie jego obiekty zależne zostają o tym powiadomione i automatycznie zaktualizowane. We wzorcu udział biorą udział dwa typy obiektów:

PrzedmiotObserwowany i Obserwator. Odpowiedzialność klasy PrzedmiotObserwowany polega na przechowywaniu listy Obserwatorów, ich dodawaniu (metoda Podlacz) i usuwaniu (metoda Odlacz), a także ich powiadamianiu o zmianach (metoda Zawiadam). Obserwator posiada interfejs umożliwiający zareagowanie na zmiany w obiekcie klasy PrzedmiotObserwowany.

Dzięki zastosowaniu wzorca Obserwator ograniczamy ilość powiązań i zależności pomiędzy obserwatorami i przedmiotem obserwowanym.

Przykładowe rozwiązanie

Definicja i wykorzystanie zdarzenia

Naszym zadaniem jest stworzenie klasy symulującej działanie sejf. W wypadku nieuprawnionego dostępu do sejf. zostanie wszczęty alarm. Działania podjęte w wyniku uruchomienia alarmu nie zależą od wykonawców sejf. – oni tylko dostarczają odpowiedni interfejs w postaci zdarzenia. Co zostanie uruchomione w odpowiedzi na zdarzenie, zależy od właściciela sejf. i zasobności jego portfela. Może on wynająć firmę ochroniarską, spuścić kratę, wypuścić psy itp. Sejf pełni niejako rolę przedmiotu obserwowanego, a psy, kraty, firma ochroniarska rolę obserwatorów.

Utworzenie delegacji

W bibliotece .NET Framework do definicji zdarzeń używa się najczęściej delegacji, które reprezentują funkcje dwuargumentowe zwracające typ void. Pierwszy argument jest typu object i zawiera informacje, na jakim obiekcie zostało wywołane zdarzenie. Drugi argument jest typu EventArgs lub klasy pochodnej po tym typie i zawiera różne informacje związane ze zdarzeniem. Przyjęto również konwencję, że delegacja ma nazwę *NazwaEventHandler*, a nazwa klasy drugiego argumentu ma postać *NazwaEventArgs*. Stosując powyższą konwencję utworzymy naszą delegację. Wywołując zdarzenie będziemy dodatkowo przysyłać informacje na temat daty i czasu, kiedy zostało ono zgłoszone. Przykładowa definicja klasy zawierającej informacje o zdarzeniu i samej delegacji znajduje się poniżej:

```
public class AlarmEventArgs: EventArgs {
    private DateTime kiedy = DateTime.Now;
    public DateTime Kiedy {
        get { return kiedy; }
    }
}

public delegate void AlarmEventHandler(object sender, AlarmEventArgs args);
```

Utworzenie klasy zawierającej zdarzenie

Z perspektywy naszego przykładu w klasie Sejf najważniejsze jest zdarzenie określone przez delegację AlarmEventHandler oraz metoda, która próbuje uzyskać dostęp do sejf. Metoda ta przyjmuje jako argument numer PIN. W przypadku, gdy jest on nieprawidłowy, wywoływane jest zdarzenie. Pamiętajmy, aby zawsze przed wywołaniem zdarzenia sprawdzić, czy są do niego podłączone jakieś funkcje. Uproszczony na potrzeby tego przykładu kod klasy Sejf może wyglądać następująco:

```
class Sejf {
    private string czyj;
    public string Czyj {
        get { return czyj; }
    }

    private int pin;
    public event AlarmEventHandler Alarm;
```

```
public Sejf(string czyj, int pin) {
    this.czyj = czyj;
    this.pin = pin;
}

public void OtworzSejf(int pin) {
    if (this.pin != pin) {
        Console.WriteLine("Zły pin. Włączamy alarm.");
        if (Alarm != null)
            Alarm(this, new AlarmEventArgs());
        return;
    }
    Console.WriteLine("Uzyskałeś dostęp do sejfu");
}
}
```

Utworzenie metod, które można podłączyć pod zdarzenie

Załóżmy, że właściciel sejfu jest bogaty i stać go na wynajęcie osobistego strażnika. Utwórzmy klasę, która będzie reprezentować strażnika. Klasa będzie posiadać metodę zgodną z definicją naszej delegacji. Wewnątrz tej metody sprawdzimy, czyj sejf uruchomił alarm oraz kiedy to nastąpiło:

```
class Straznik {
    private string imie, nazwisko;
    public Straznik(string imie, string nazwisko) {
        this.imie = imie;
        this.nazwisko = nazwisko;
    }

    public void GonZlodzieja(object sender, AlarmEventArgs e) {
        Sejf s = (Sejf)sender;
        Console.WriteLine("Strażnik {0} {1} przyjął zgłoszenie włączenia" +
            " alarmu do sejfu pana(i) {2}. Alarm został włączony {3}.",
            imie, nazwisko, s.Czyj, e.Kiedy.ToString());
    }
}
```

Drugim sposobem zabezpieczenie sejfu będzie podłączenie syreny. W przypadku nieuprawnionego dostępu do sejfu syrena zacznie wyć. Metoda, którą zaraz zdefiniujemy, w odróżnieniu od metody `GonZlodzieja` z przykładu powyżej, będzie metodą statyczną:

```
static class Syrena {
    public static void Wyj(object sender, AlarmEventArgs e) {
        for (int i = 0; i < 100; i++) {
            Console.Beep(1000 + 100 * i, 10);
        }
    }
}
```

Utworzenie programu testującego

W celu przetestowania działania zdarzeń utworzymy dwa sejfy, kilku strażników i podłączymy odpowiednie metody pod zdarzenie `Alarm`. Następnie spróbujemy uzyskać dostęp do sejfu podając zły PIN:

```
Sejf sejf1 = new Sejf("Bill G.", 1234);
Sejf sejf2 = new Sejf("Jan K.", 4321);

Straznik s1 = new Straznik("Chuck", "Norris");
Straznik s2 = new Straznik("John", "Rambo");
```

```
sejf1.Alarm += new AlarmEventHandler(Syrena.Wyj);  
sejf1.Alarm += new AlarmEventHandler(s1.GonZlodzieja);  
sejf2.Alarm += new AlarmEventHandler(s1.GonZlodzieja);  
sejf2.Alarm += new AlarmEventHandler(s2.GonZlodzieja);  
  
sejf1.OtworzSejf(5678);  
Console.ReadKey(true);  
  
sejf2.OtworzSejf(8765);  
Console.ReadKey(true);
```

W wyniku działania powyższego kodu, oprócz tego że usłyszysz sygnał dźwiękowy (oczywiście pod warunkiem, że masz podłączony głośnik), na ekranie będziesz mógł zobaczyć następujący tekst (naturalnie data i godzina mogą być inne):

```
Zły pin. Włączamy alarm.  
Strażnik Chuck Norris przyjął zgłoszenie włączenia alarmu do sejfu pana(i)  
Bill G.. Alarm został włączony 2009-03-26 14:15:17.
```

```
Zły pin. Włączamy alarm.  
Strażnik Chuck Norris przyjął zgłoszenie włączenia alarmu do sejfu pana(i)  
Jan K.. Alarm został włączony 2009-03-26 14:15:20.  
Strażnik John Rambo przyjął zgłoszenie włączenia alarmu do sejfu pana(i)  
Jan K.. Alarm został włączony 2009-03-26 14:15:20.
```

Pan Jan K. może się zdenerwować, że jeden ze strażników ma drugą chałturę i go zwolnić. Odłączenie jednej z metod od zdarzenia można wykonać następująco:

```
sejf2.Alarm -= new AlarmEventHandler(s1.GonZlodzieja);  
sejf2.OtworzSejf(8765);
```

W wyniku działania powyższego kodu zgłosi się już tylko jeden strażnik.

Gotowy rozwiązany powyższy przykład znajduje się w katalogu **Demo\Modul12**.

Porady praktyczne

- Delegacje są podobne do wskaźników na funkcję z innych języków programowania (np. C/C++), jednak w odróżnieniu od nich są w pełni obiektowe oraz bezpieczne ze względu na typ.
- Do definicji delegacji, która reprezentuje metody nie zwracające żadnej wartości (zwracające typ `void`), możesz użyć delegacji predefiniowanej `System.Action`. Istnieje pięć wersji tej delegacji: wersja, która reprezentuje metody bezparametrowe oraz cztery wersje generyczne, które odpowiednio przyjmują jeden, dwa, trzy lub cztery argumenty. Oczywiście typy parametrów delegacji generycznej są określone przez parametry typu.
- Do definicji delegacji, która reprezentuje metody zwracające jakąś wartość (tj. takie, których typ wartości zwracanej nie jest określony przez typ `void`), możesz użyć delegacji predefiniowanej `System.Func`. Istnieje pięć generycznych wersji tej delegacji: wersja, która reprezentuje metody bezparametrowe oraz wersje, które odpowiednio przyjmują jeden, dwa, trzy lub cztery argumenty. Oczywiście typ wartości zwracanej i typy parametrów delegacji reprezentujących metody z parametrami są określone przez parametry typu. Parametr typu reprezentujący wartość zwracaną występuje jako ostatni na liście parametrów typu.
- Do definicji delegacji reprezentującej metody sprawdzające, czy obiekt danego typu spełnia pewne kryteria (tj. posiadające jeden parametr i zwracające wartość typu logicznego `bool`) możesz wykorzystać predefiniowaną delegację generyczną `System.Predicate<T>`.

- Obiekt delegacji, przy pomocy którego chcesz wywołać asynchronicznie metodę, może reprezentować tylko pojedynczą metodę.
- Tworząc nową delegację, która będzie używana do definicji zdarzeń, stosuj następującą konwencję:
 - Delegacja powinna reprezentować metody, które przyjmują dwa argumenty i zwracają typ `void`. Pierwszy argument jest typu `object` i zawiera informację, na jakim obiekcie wywołane zostało zdarzenie. Drugi argument jest klasą pochodną po typie `EventArgs` i zawiera różne informacje związane ze zdarzeniem.
 - Delegacja posiada nazwę *NazwaEventHandler*, gdzie przedrostek *Nazwa* jest dowolnym identyfikatorem sugerującym, do definicji jakiego rodzaju zdarzeń dana delegacja może zostać wykorzystana.
 - Nazwa klasy drugiego argumentu ma postać *NazwaEventArgs*, gdzie przedrostek *Nazwa* jest dowolnym identyfikatorem.
- Definiując delegację, którą będziesz wykorzystywał(a) do definicji zdarzeń, możesz wykorzystać delegację generyczną `System.EventHandler<TEventArgs>`. Na parametr typu `TEventArgs` nałożone jest ograniczenie, że musi dziedziczyć po typie `EventArgs`.
- Przed wywołaniem zdarzenia sprawdź, czy pod zdarzenie podłączona jest jakaś metoda.
- Definiując metodę która będzie obsługiwać zdarzenie, pierwszy jej argument nazwij `sender`, drugi natomiast – ten, który zawiera dodatkowe informacje związane ze zdarzeniem – nazwij `e` lub `args`.
- W bibliotece .NET Framework stosowana jest pewna konwencja, którą możesz również stosować w swoich klasach. Z wieloma czynnościami mogą być skojarzone dwa zdarzenia:
 - Zdarzenie wywoływane przed wykonaniem danej czynności. Nazwa tego zdarzenia ma końcówkę `-ing`, np. `Validating` czy `FormClosing`. Jako drugi argument delegacji służącej do definicji tego zdarzenia, przesyłany jest obiekt typu `System.ComponentModel.CancelEventArgs` lub klasy pochodnej po tym typie. Obiekty klasy `CancelEventArgs` posiadają właściwość `Cancel`, przy pomocy której możemy anulować wykonanie czynności, której rozpoczęcie spowodowało wywołanie danego zdarzenia.
 - Zdarzenie wywoływane po wykonaniu danej czynności. Nazwa tego zdarzenia ma końcówkę `-ed`, np. `Validated` czy `FormClosed`.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- wiesz i rozumiesz, co to jest delegacja i potrafisz ją definiować
- wiesz, co to są wyrażenia `lambda`
- potrafisz definiować funkcje przy pomocy wyrażenia `lambda`
- potrafisz podłączać metody pod delegację (zarówno metodę instancji, jak i metodę statyczną)
- wiesz jak wywołać metody skojarzone z danym obiektem delegacji
- wiesz jak wywołać delegację asynchronicznie
- wiesz i rozumiesz, co to jest zdarzenie
- potrafisz definiować własne zdarzenia i je wywoływać
- rozumiesz wzorzec projektowy Obserwator

Dodatkowe źródła informacji

1. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka skierowana do programistów chcących nauczyć się programować w języku C#.

2. Andrew Troelsen, *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition*, Apress, 2007

Książka przeznaczona dla bardziej zaawansowanych programistów. Czwarte wydanie opisuje język C# 3.0 i platformę .NET 3.5.

3. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad jak powinniśmy pisać swoje programy.

4. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Wzorce projektowe Wydanie II*, WNT, 2008

Książka, która wprowadziła pojęcie wzorców projektowych do informatyki. Lektura obowiązkowa dla każdego, kto chce poznać temat wzorców projektowych.

5. Steven John Metsker, *C#. Wzorce projektowe*, Helion, 2005

Książka jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia, jak wykorzystać cechy języka C# do tworzenia poprawnego kodu poprzez zastosowanie wzorców.

6. Judith Bishop, *C# 3.0 Design Patterns*, O'Reilly Media, Inc. 2008

Książka, podobnie jak poprzednia, jest przewodnikiem po wzorcach projektowych w C# i środowisku .NET. Przedstawia również nowe cechy języka C#.

7. Codeguru, <http://www.codeguru.pl>

Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.

8. C Sharp Tutorial, http://www.meshplex.org/wiki/C_Sharp_Tutorial

Internetowy kurs języka C#.

9. C# Corner, <http://www.csharpcorner.com>

Portal poświęcony programowaniu w języku C#.

10. Kurs C#, cz. I, http://www.centrumxp.pl/dotNet/20,1,kategoria,Kurs_C_cz_I.aspx

Przystępny kurs języka C# w języku polskim.

11. MSDN Library, <http://msdn.microsoft.com>

Na szczególną uwagę, oprócz opisu słów kluczowych event i delegate oraz opisu poszczególnych typów delegacji, zasługuje artykuł techniczny: Doug Purdy, Jeffrey Richter, *Exploring the Observer Design Pattern*.

Laboratorium podstawowe

Problem 1 (czas realizacji 15 min)

W firmie potrzebne są funkcje, które wykonują pewne obliczenia statystyczne na kolekcjach liczb rzeczywistych. Niestety przy obliczeniach nie wszystkie elementy kolekcji mogą być wzięte pod uwagę. Kryterium, które elementy należy użyć do obliczenia wartości funkcji statystycznej, zmienia się w zależności od potrzeb. Jesteś szczęśliwcem, który otrzymał za zadanie napisania biblioteki takich funkcji. Zaimplementuj metody służące do wyznaczania:

- liczby elementów kolekcji spełniających dane kryterium
- sumy elementów kolekcji spełniających dane kryterium
- średniej z elementów kolekcji spełniających dane kryterium
- wartości maksymalnej z wartości elementów kolekcji spełniających dane kryterium
- wartości minimalnej z wartości elementów kolekcji spełniających dane kryterium

Uwaga:

Tak naprawdę wszystkie powyższe metody są już zaimplementowane w bibliotece FCL, przy pomocy metod rozszerzających interfejs `IEnumerable<T>`.

Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition typu ClassLibrary	<ul style="list-style-type: none"> • Otwórz Visual C# 2008 Express Edition. • Z menu wybierz File -> New Project. • Z listy Visual Studio installed templates wybierz Class Library. • W polu Name wpisz Statystyka. • Kliknij OK. • Z menu wybierz File -> Save Statystyka. • W polu Location wybierz folder w którym będzie zapisany projekt. • Zaznacz pole wyboru Create directory for solution. • W polu Solution Name wpisz Modul12. • Naciśnij przycisk Save.
2. Utwórz statyczną klasę FunkcjeStatystyczne, która będzie zawierać żądane metody	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na FunkcjeStatystyczne.cs. W okienku dialogowym naciśnij przycisk Tak. • Uczyń klasę Zestaw klasą statyczną: <pre>public static class FunkcjeStatystyczne { }</pre>
3. Do klasy FunkcjeStatystyczne dodaj metodę wyznaczającą ilość elementów kolekcji spełniających kryterium	<ul style="list-style-type: none"> • Do klasy FunkcjeStatystyczne dodaj statyczną metodę wyznaczającą liczbę elementów kolekcji spełniających zadane kryterium: <pre>public static int Ilosc(IEnumerable<double> tab, Func<double, bool> kryterium) { int i = 0; foreach (double d in tab) { if (kryterium(d)) { i++; } } return i; }</pre>
4. Do klasy FunkcjeStatystyczne	<ul style="list-style-type: none"> • Do klasy FunkcjeStatystyczne dodaj statyczną metodę wyznaczającą sumę elementów kolekcji spełniających zadane kryterium:

e dodaj metodę wyznaczającą sumę elementów kolekcji spełniających kryterium	<pre> public static double Suma(IEnumerable<double> tab, Func<double, bool> kryterium) { double suma = 0; foreach (double d in tab) { if (kryterium(d)) { suma += d; } } return suma; } </pre>
5. Do klasy FunkcjeStatystyczne e dodaj metodę wyznaczającą średnią z elementów kolekcji spełniających kryterium	<ul style="list-style-type: none"> Do klasy FunkcjeStatystyczne dodaj statyczną metodę wyznaczającą średnią z elementów kolekcji spełniających zadane kryterium: <pre> public static double SredniaArytmetyczna(IEnumerable<double> tab, Func<double, bool> kryterium) { double suma = 0; int i = 0; foreach (double d in tab) { if (kryterium(d)) { suma += d; i++; } } if (i == 0) { throw new ArgumentException("Żaden element kolekcji nie " + "spełnia kryterium"); } return suma / i; } </pre>
6. Do klasy FunkcjeStatystyczne e dodaj metodę wyznaczającą maksimum z elementów kolekcji spełniających kryterium	<ul style="list-style-type: none"> Do klasy FunkcjeStatystyczne dodaj statyczną metodę wyznaczającą maksimum elementów kolekcji spełniających zadane kryterium: <pre> public static double Maksimum(IEnumerable<double> tab, Func<double, bool> kryterium) { bool znaleziono = false; double max = double.MinValue; foreach (double f in tab) { if (kryterium(f) && (f > max !znaleziono)) { max = f; znaleziono = true; } } if (!znaleziono) { throw new InvalidOperationException("Nie znaleziono żadnego " + "elementu, który spełnia zadane kryterium."); } return max; } </pre>
7. Do klasy FunkcjeStatystyczne e dodaj metodę wyznaczającą minimum z elementów kolekcji spełniających kryterium	<ul style="list-style-type: none"> Do klasy FunkcjeStatystyczne dodaj statyczną metodę wyznaczającą minimum elementów kolekcji spełniających zadane kryterium: <pre> public static double Minimum(IEnumerable<double> tab, Func<double, bool> kryterium) { bool znaleziono = false; double min = double.MaxValue; foreach (double f in tab) { if (kryterium(f) && (f < min !znaleziono)) { min = f; znaleziono = true; } } if (!znaleziono) { throw new InvalidOperationException("Nie znaleziono żadnego " + "elementu, który spełnia zadane kryterium."); } } </pre>

	<pre> } return min; return min; } </pre>
8. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestFunkcjiStatystycznych. Kliknij OK.
9. Zaznacz projekt TestFunkcjiStatystycznych jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestFunkcjiStatystycznych, a następnie z menu kontekstowego wybierz Set as StartUp Project.
10. Do projektu TestFunkcjiStatystycznych dodaj odwołanie do biblioteki Statystyka	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projektu TestFunkcjiStatystycznych, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt Statystyka, a następnie kliknij przycisk OK.
11. Zaimplementuj metodę Main	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Na górze pliku Program.cs zaimportuj przestrzeń nazw Statystyka: <pre>using Statystyka;</pre> Do metody Main dodaj kod, który przetestuje wcześniej utworzone metody: <pre> static void Main(string[] args) { double[] tab = { 2, 3, 6, 9 }; Console.WriteLine("Liczba elementów tablicy większych od " + "pięciu: {0}", FunkcjeStatystyczne.Ilosc(tab, p => p > 5)); Console.WriteLine("Suma elementów tablicy większych od " + "pięciu: {0}", FunkcjeStatystyczne.Suma(tab, p => p > 5)); Console.WriteLine("Średnia elementów tablicy większych od " + "pięciu: {0}", FunkcjeStatystyczne.SredniaArytmetyczna(tab, p => p > 5)); Console.WriteLine("Największy z elementów tablicy większych " + "od pięciu: {0}", FunkcjeStatystyczne.Maksimum(tab, p => p > 5)); Console.WriteLine("Najmniejszy z elementów tablicy większych " + "od pięciu: {0}", FunkcjeStatystyczne.Minimum(tab, p => p > 5)); Console.ReadKey(); } </pre>
12. Skompiluj i uruchom program	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Problem 2 (czas realizacji 30 min)

W firmie opracowujecie pewien program. Jednym z jego składników jest system bezpieczeństwa. Twoim zadaniem jest stworzenie klasy reprezentującej konto, zawierającej informacje na temat nazwy użytkownika i hasła. Dodatkowo Twoja klasa powinna udostępniać:

- Możliwość zmiany hasła.
- Możliwość sprawdzenia przed zmianą hasła, czy nowe hasło spełnia pewne kryteria. W przypadku gdy hasło nie spełnia żądanych kryteriów, operacja zmiany hasła powinna być anulowana. Kryteria określające, czy hasło jest poprawne, będą definiowane poza klasą reprezentującą konto. Do wymagań wobec hasła może należeć np. jego długość czy stopień skomplikowania.
- Możliwość definicji operacji wykonywanej w przypadku, gdy hasło udało się zmienić, np. w celu inspekcji. Czynności wykonywane po zmianie hasła (np. zapis informacji o zmianie hasła do pliku) również będą definiowane na zewnątrz klasy reprezentującej konto. Programiści korzystający z Twojej klasy w przyszłości będą dostarczać metody wykonujące żądane w danym przypadku operacje.

Napisz również program testujący Twoją klasę.

Zadanie	Tok postępowania
1. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> • W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. • Z listy Visual Studio installed templates wybierz Class Library. • W polu Name wpisz Konta. • Kliknij OK.
2. Utwórz klasę Konto	<ul style="list-style-type: none"> • W okienku Solution Explore zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na Konto.cs. W okienku dialogowym naciśnij przycisk Tak.
3. Utwórz delegację, która będzie używana do definicji zdarzenia wywołanego przed zmianą hasła	<ul style="list-style-type: none"> • W pliku Konto.cs zaimportuj przestrzeń nazw System.ComponentModel. <code>using System.ComponentModel;</code> • W przestrzeni nazw Konta utwórz klasę, której obiekty będą zawierać informacje związane ze zdarzeniem wywołanym przed zmianą hasła. Informacje, które będą potrzebne, to nowe i stare hasło. Klasa ta powinna dziedziczyć po klasie System.ComponentModel.CancelEventArgs: <pre>public class PrzedZmianaHaslaArgs : CancelEventArgs { private string noweHaslo; public string NoweHaslo { get { return noweHaslo; } } private string stareHaslo; public string StareHaslo { get { return stareHaslo; } } public PrzedZmianaHaslaArgs(string noweHaslo, string stareHaslo) { this.noweHaslo = noweHaslo; this.stareHaslo = stareHaslo; } }</pre> • W przestrzeni nazw Konta utwórz delegację, która będzie używana do definicji zdarzenia wywołanego przed zmianą hasła: <pre>public delegate void PrzedZmianaHaslaHandler(object sender, PrzedZmianaHaslaArgs e);</pre>

4. Zaimplementuj klasę Konto	<ul style="list-style-type: none"> Do klasy Konto dodaj prywatne pole reprezentujące aktualne hasło: <code>private string haslo;</code> Do klasy Konto dodaj publiczną metodę SprawdzHaslo, która będzie sprawdzać poprawność hasła: <pre>public bool SprawdzHaslo(string haslo) { if (this.haslo == haslo) return true; return false; }</pre> Do klasy Konto dodaj publiczną właściwość NazwaUzytkownika, reprezentującą nazwę użytkownika: <code>public string NazwaUzytkownika { set; get; }</code> Do klasy Konto dodaj publiczny konstruktor, do którego będziemy przysyłać nazwę użytkownika oraz hasło: <pre>public Konto(string nazwaUzytkownika, string haslo) { NazwaUzytkownika = nazwaUzytkownika; this.haslo = haslo; }</pre> Do klasy Konto dodaj publiczne statyczne zdarzenie PrzedZmianaHaslaHandler, wywoływane przed zmianą hasła: <code>public static event PrzedZmianaHaslaHandler PrzedZmianaHasla;</code> Do klasy Konto dodaj publiczne statyczne zdarzenie PoZmianieHasla, wywoływane po zmianie hasła: <code>public static event EventHandler PoZmianieHasla;</code> Do klasy Konto dodaj publiczną metodę, przy pomocy której będziesz mógł zmienić hasło danego użytkownika: <pre>public bool ZmienHaslo(string stareHaslo, string noweHaslo) { if (!SprawdzHaslo(stareHaslo)) return false; if (PrzedZmianaHasla != null) { PrzedZmianaHaslaArgs e = new PrzedZmianaHaslaArgs(noweHaslo, stareHaslo); PrzedZmianaHasla(this, e); if (e.Cancel) return false; } haslo = noweHaslo; if (PoZmianieHasla != null) PoZmianieHasla(this, EventArgs.Empty); return true; }</pre>
5. Skompiluj projekt	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program.
6. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz TestKonta.

	<ul style="list-style-type: none"> Kliknij OK.
7. Zaznacz projekt TestKonta jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt TestKonta, a następnie z menu kontekstowego wybierz Set as StartUp Project.
8. Do projektu TestKonta dodaj odwołanie do biblioteki Konta	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projektu TestKonta, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt Konta, a następnie kliknij przycisk OK. Przejdź do pliku Program.cs. Na górze pliku Program.cs zaimportuj przestrzeń nazw Konta: <pre>using Konta;</pre>
9. Dodaj klasę, której metody będą wywoływane przed i po zmianie hasła	<ul style="list-style-type: none"> Na górze pliku Program.cs zaimportuj przestrzeń nazw System.IO: <pre>using System.IO;</pre> W przestrzeni nazw TestKonta dodaj statyczną klasę PolitykaBezpieczenstwa: <pre>static class PolitykaBezpieczenstwa { }</pre> Do klasy dodaj publiczną statyczną właściwość reprezentującą minimalną liczbę znaków hasła: <pre>public static int MinimalnaLiczbaZnakow { set; get; }</pre> Do klasy PolitykaBezpieczenstwa dodaj publiczną statyczną właściwość reprezentującą nazwę pliku, w którym będą zapisywane informacje o zmianach haseł: <pre>public static string NazwaPliku { set; get; }</pre> Do klasy PolitykaBezpieczenstwa dodaj statyczny konstruktor, który będzie ustawiał nazwę pliku i minimalną liczbę znaków hasła: <pre>static PolitykaBezpieczenstwa() { MinimalnaLiczbaZnakow = 5; NazwaPliku = "Audyty.txt"; }</pre> Do klasy PolitykaBezpieczenstwa dodaj publiczną statyczną metodę sprawdzającą, czy nowe hasło ma odpowiednią liczbę znaków. Sygnatura metody powinna być zgodna z delegacją PrzedZmianaHaslaHandler: <pre>public static void SprawdzDlogoscHasla(object sender, PrzedZmianaHaslaArgs e) { if (e.NoweHaslo.Length < MinimalnaLiczbaZnakow) e.Cancel = true; }</pre> Do klasy PolitykaBezpieczenstwa dodaj publiczną statyczną metodę sprawdzającą, czy nowe hasło nie jest takie samo jak poprzednie. Sygnatura metody powinna być zgodna z delegacją PrzedZmianaHaslaHandler: <pre>public static void SprawdzCzyHasloJestPowtorzone(object sender, PrzedZmianaHaslaArgs e) { if (e.NoweHaslo == e.StareHaslo) e.Cancel = true; }</pre> Do klasy PolitykaBezpieczenstwa dodaj publiczną statyczną metodę zapisującą informację o zmianie hasła do pliku. Sygnatura metody

	<p>powinna być zgodna z delegacją EventHandler:</p> <pre> public static void ZapiszDoPliku(object sender, EventArgs e) { StreamWriter sw = null; try { sw = new StreamWriter(NazwaPliku, true); Konto k = (Konto)sender; sw.WriteLine("Zmiana hasła dla użytkownika {0}", k.NazwaUzytkownika); } finally { if (sw != null) sw.Close(); } } </pre>
10. Zaimplementuj metodę Main	<ul style="list-style-type: none"> Przetestuj działanie klasy Konto korzystając z klasy PolitykaBezpieczeństwa. Przykładowy kod jest umieszczony poniżej: <pre> static void Main(string[] args) { Konto k1 = new Konto("JanKowalski", "password"); Konto k2 = new Konto("AndrzejNowak", "haslo"); Konto.PrzedZmianaHasla += new PrzedZmianaHaslaHandler(PolitykaBezpieczeństwa.SprawdxCzyHasloJestPowtorzone); Konto.PrzedZmianaHasla += new PrzedZmianaHaslaHandler(PolitykaBezpieczeństwa.SprawdzDlogoscHasla); Konto.PoZmianieHasla += new EventHandler(PolitykaBezpieczeństwa.ZapiszDoPliku); if (k1.ZmienHaslo("password", "123")) Console.WriteLine("Hasło zmienione."); else Console.WriteLine("Hasło nie udało się zmienić."); if (k2.ZmienHaslo("haslo", "haslo")) Console.WriteLine("Hasło zmienione."); else Console.WriteLine("Hasło nie udało się zmienić."); if (k1.ZmienHaslo("password", "12345")) Console.WriteLine("Hasło zmienione."); else Console.WriteLine("Hasło nie udało się zmienić."); if (k2.ZmienHaslo("haslo", "123haslo")) Console.WriteLine("Hasło zmienione."); else Console.WriteLine("Hasło nie udało się zmienić."); Console.ReadKey(); } </pre>
11. Skompiluj i uruchom program	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. W celu uruchomienia programu z menu Debug wybierz Start Debugging.

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 90 min)

Firma, w której pracujesz, dostała zlecenie na napisanie systemu sterowania. Jednym z podsystemów tej aplikacji jest system planowania i zarządzania zadaniami. Ponieważ kolejnym zleceniem będzie napisanie zaawansowanego osobistego organizera, który będzie między innymi przypominał o spotkaniach, zarząd firmy postanowił stworzyć uniwersalną klasę, przy pomocy której będzie można uruchamiać pewne funkcje o ustalonej godzinie. Podczas burzliwych narad ustalono następujące wymagania co do tej klasy:

- Obiekty tworzonej klasy mogą przechowywać dowolną liczbę zadań do wykonania.
- Każde zadanie ma nazwę.
- Termin wykonania zadania może być określony przez podanie dnia i godziny, kiedy zadanie ma się rozpocząć wykonywać lub ilości minut, za ile zadanie ma się rozpocząć wykonywać.
- Użytkownik może definiować zadania, które będą uruchamiane jednorazowo lub cyklicznie. Przy definicji zadań cyklicznych należy podać, co ile minut należy ponownie uruchomić zadanie.
- Metody, które realizują konkretne zadania, powinny być uruchamiane asynchronicznie.
- Informacje o sukcesie, ewentualnie porażce realizacji danego zadania powinny być zapisane w pojedynczym pliku logu. Informacje, które powinny być tam zapisane to: nazwa zadania, data i godzina rozpoczęcia zadania oraz data i godzina zakończenia zadania.
- Użytkownik może uzyskać listę zadań, które zostały zaplanowane dla danego obiektu implementowanej klasy. Informacje, które może uzyskać to:
 - nazwa zadania
 - data i godzina rozpoczęcia wykonywania zadania (planowana lub już wykonana)
 - informacje o statusie zadania: oczekujące, aktualnie wykonywane, wykonane z sukcesem, wykonane z błędem
 - dodatkowo w przypadku zadań wykonywanych cyklicznie: co ile minut zadanie jest powtarzane, ile razy rozpoczęto wykonywać zadanie, ile razy wykonanie zadania zakończyło się sukcesem

Ponieważ planowane aplikacje mają pracować pod systemem Windows, do implementacji żądanej klasy wybrano język C#. Ty w firmie jesteś najlepszym specjalistą od Platformy .NET i języka C#, więc utworzenie wymaganej klasy przypadło Tobie w udziale.

Uwaga: w programie możesz użyć klasy `System.Threading.Timer` do wywoływania cyklicznie funkcji, w której będziesz sprawdzał, czy istnieją jakieś zadania, które powinny być uruchomione.

ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 13

Wersja 2

Refleksja i atrybuty

Spis treści

Refleksja i atrybuty	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	8
Porady praktyczne	11
Uwagi dla studenta	12
Dodatkowe źródła informacji	12
Laboratorium podstawowe	14
Laboratorium rozszerzone	20
Zadanie 1 (czas realizacji 90 min)	20

Informacje o module

Opis modułu

W tym module zapoznasz się z pojęciem refleksji. Dowiesz się, jak korzystać z tego mechanizmu w języku C#, a także jak dodawać dodatkowe informacje (metadane) do programu przy pomocy atrybutów. Nauczysz się również definiować własne atrybuty.

Cel modułu

Celem modułu jest przedstawienie mechanizmu uzyskiwania informacji o typach w czasie działania programu oraz omówienie atrybutów i sposobu ich wykorzystania w języku C#.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- wiedział, co to jest refleksja
- potrafił uzyskać informacje o typie w czasie działania programu
- wiedział, co to są atrybuty
- umiał definiować własne atrybuty i z nich korzystać

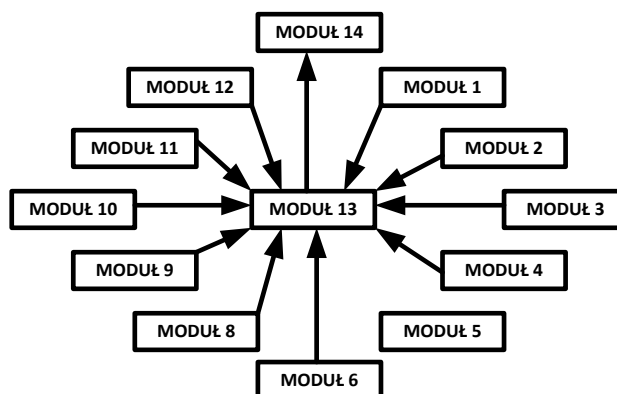
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinienś:

- rozumieć pojęcie dziedziczenia
- rozumieć pojęcie właściwości
- potrafić definiować własne właściwości
- rozumieć pojęcie interfejsu

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w modułach „Pojęcie klasy”, „Właściwości i indeksatory”, „Składowe statyczne”, „Dziedziczenie”, „Polimorfizm – funkcje wirtualne”, „Klasy abstrakcyjne i interfejsy”, „Typy ogólne”, „Implementacja przykładowych interfejsów” oraz „Delegacje i zdarzenia”.



Rys. 20 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Otrzymałeś ostatnio zadania napisanie programu. Szef przedstawił Ci główne założenia co do funkcjonalności. Następnie stwierdził, że program powinien być łatwo rozszerzalny. Gdy zapytałeś, o co mu dokładnie chodzi odpowiedział, że podczas uruchamiania program powinien sprawdzić pewien katalog, wczytać z niego po kolei wszystkie biblioteki DLL (zestawy) i sprawdzić, czy zawierają klasy, które mogą rozszerzyć jego funkcjonalność. Po znalezieniu rozszerzeń, program powinien zapytać użytkownika, czy chce je dodać do programu, a następnie w przypadku twierdzącej odpowiedzi zastosować te rozszerzenia. Gdy zapytałeś czy chodzi mu o mechanizm wtyczek – potwierdził. Szef również dodał, że dobry by było, gdyby każda z klas posiadała informację na temat tego, kto jest jej autorem, numer wersji lub ewentualnie datę utworzenia. Pomyślałeś chwilę i stwierdziłeś że w celu uzyskania informacji o klasie utworzysz interfejs, który będzie posiadał metody `GetName`, `GetDate` oraz `GetVersion` i który będzie implementowany przez wszystkie klasy rozszerzające Twoją aplikację. Szef dziwnie popatrzył na Ciebie i zapytał, czy nie słyszałeś o atrybutach. Wolałeś nic nie odpowiadać. Gdy mu powiedziałaś, że nie wiesz za bardzo, jak wczytać podzespół i wyszukać w nich odpowiednich klas, najpierw się zaczerwienił, wykrzyczał kilka pojęć takich jak refleksja, późne wiązanie, dynamicznie ładowany podzespół czy klasa `Activator`, a następnie wręczył Ci ten oto moduł życząc miłej lektury.

Podstawy teoretyczne

Refleksja w języku C#, a tak naprawdę w całym .NET Framework, jest to uzyskiwanie informacji o typie w czasie działania programu. Przy pomocy tego mechanizmu możemy otrzymać podobne informacje, jakie wyświetla narzędzie `ILDasm` (deassembler kodu pośredniego) dostarczane razem z .NET Framework czy program `Reflector .NET` (do pobrania z Internetu¹).

Informacje o typie są niejako opakowane przez klasę `System.Type`. Klasa ta zawiera szereg właściwości (np.: `IsAbstract`, `IsArray`, `IsEnum`) oraz metod (np. `GetConstructors`, `GetEvents` czy `FindMembers`), których nie będziemy tutaj szczegółowo opisywać, ponieważ nazwy dostatecznie dobrze oddają ich przeznaczenie. Przykładowo metody klasy `Type` o nazwie rozpoczynającej się od przedrostka `Get` zwracają obiekty, których typ jest określony przez jedną z klas znajdujących się w przestrzeni nazw `System.Reflection`, np. `EventInfo`, `FieldInfo` czy `MemberInfo` i tak metoda `GetEvents` klasy `Type` zwraca tablicę obiektów typu `EventInfo`. W razie wątpliwości można zawsze zajrzeć do dokumentacji.

Obiektu klasy `Type` nie możemy utworzyć bezpośrednio przy pomocy słowa kluczowego `new`, ponieważ `Type` jest klasą abstrakcyjną. Mimo tego w dalszej części tego modułu będziemy posługiwać się pojęciem obiektu typu `Type` pamiętając, że tak naprawdę jest to obiekt jednej z klas pochodnych (są to klasy wewnętrzne, a ich nazwy rozpoczynają się od słowa „Runtime”). Wracając do meritum sprawy, mamy wiele sposobów otrzymania obiektu typu `Type`. Omówimy tutaj trzy, chyba najpopularniejsze:

- metoda `GetType` typu `object`
- operator `typeof`
- metoda statyczna `GetType` klasy `Type`

Przykład użycia metody `Object.GetType` zastał umieszczony poniżej:

```
MojaKlasa obj = new MojaKlasa();
Type t = obj.GetType();
```

¹ <http://www.red-gate.com/products/reflector/>

Powyższy sposób przydaje się, gdy chcemy zasięgnąć informacji o typie jakiegoś obiektu. Zanim z niego skorzystamy, musimy zawsze utworzyć obiekt danego typu. Problem ten rozwiązuje operator `typeof`, co pokazuje następujący przykład:

```
Type t = typeof(nazwa_typu);
```

Wprawdzie przy pomocy operatora `typeof` nie trzeba tworzyć obiektu, ale nazwa typu, o którym chcemy zasięgnąć informacji, musi być znana w czasie kompilacji. Najbardziej uniwersalna wydaje się metoda `GetType` klasy `Type`. Do metody tej nazwę typu przekazujemy w postaci napisu. Użycie metody `GetType` demonstruje poniższy kod:

```
Type t = Type.GetType("System.String");
```

Warto zwrócić uwagę, że metoda `GetType` ma przeciążoną wersję, gdzie dodatkowo możemy podać, czy w wypadku, gdy podany typ nie zostanie znaleziony rzucić wyjątek oraz czy przy wyszukiwaniu brać pod uwagę wielkość liter. Oba parametry oczywiście są typu logicznego. Powyższy sposób zadziała, gdy typ, o którym chcemy pobrać informację jest zdefiniowany w zestawie aktualnie wykonywanym lub bibliotece **mscorlib.dll**. W przypadku, gdy żądany typ jest zdefiniowany w innym zestawie, po nazwie klasy stawiamy przecinek i podajemy nazwę zestawu, co demonstruje poniższy kod:

```
Type t1 = Type.GetType("Macierze.Macierz, Macierze");  
Type t2 = Type.GetType("System.Drawing.Color," +  
    "System.Drawing,Version=2.0.0.0," +  
    "Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a");
```

Pierwszy przykład pokazuje, jak można pobrać informacje na temat typu `Macierz`, który znajduje się w przestrzeni nazw `Macierze` i jest zdefiniowanym w lokalnym zestawie **Macierze** (zestawie znajdującym się w katalogu aplikacji). Drugi przykład demonstruje pobranie informacji o typie znajdującym się w zestawie współdzielonym. Typy możemy definiować wewnątrz klasy lub struktury. W celu odwołania się do typu zagnieżdżonego, między nazwą nadrzędnej klasy lub struktury a typem zagnieżdżonym stawiamy znak plus:

```
Type.GetType("PrzestrzenNazw.Klasa+KlasaZagniezdzona,NazwaZestawu")
```

Specyficzną notację stosujemy również, gdy próbujemy pobrać informację o typach ogólnych. W tym przypadku musimy podać ilość parametryzujących typów, liczbę tę umieszczamy po znaku akcent „```”, który powinien znajdować się tuż za nazwą typu ogólnego. Po liczbie parametryzujących typów w nawiasie kwadratowym podajemy nazwy tych typów.

```
Type.GetType("System.Collections.Generic.Dictionary`2" +  
    "[System.String,[MojTyp,NazwaZestawu]]")
```

W powyższym przykładzie pokazano również co zrobić, gdy jeden z parametryzujących typów jest zdefiniowany w innym zestawie niż aktualnie wykonywany lub **mscorlib.dll**.

Dynamicznie ładowane zestawy i późne wiązanie

Korzystając z klas z jakiegoś zestawu dodajemy referencję do tego zestawu. Powoduje to, że w manifestcie naszego programu znajduje się odwołanie do danego zestawu. W .NET istnieje również możliwość ładowania zestawów w sposób dynamiczny, w czasie działania programu. W dowolnym momencie działania programu możemy wczytać zestawy z wybranych katalogów, a następnie przejrzeć typy w nich zdefiniowane pod względem pewnych kryteriów (np. implementacji interfejsów). W przypadku, gdy typ spełniający dane kryteria istnieje, możemy utworzyć jego instancję, a następnie wywołać metodę na rzecz danej instancji. Zwróćmy uwagę, że nie musimy w czasie tworzenia programu znać nazwy danego typu. W przypadku wykorzystania mechanizmu funkcji wirtualnych, musiało gdzieś w programie istnieć miejsce, gdzie w sposób jawny tworzyliśmy obiekt danej klasy. W przypadku późnego wiązania (ang. *late binding*) nazwy klasy nie trzeba podawać w sposób jawny.

Do dynamicznego ładowania zestawów możemy użyć metod statycznych `Load` lub `LoadFrom` klasy `System.Reflection.Assembly`. Metoda `LoadFrom` służy do ładowania zestawu z konkretnej lokalizacji:

```
Assembly a = Assembly.LoadFrom("c:\\Katalog\\Przykładowy_Zestaw.dll");
```

Metodą `Load` możemy wczytać zestaw o danej nazwie:

```
Assembly a = Assembly.Load(@"NazwaZestawu, Version=1.0.982.23972," +  
    @"PublicKeyToken=null, Culture=");
```

Najlepiej różnice między obiema metodami pokazuje poniższy przykład. W tym przykładzie zakładamy, że w katalogu aplikacji znajduje się zestaw o nazwie `Zestaw`, zaimplementowany w pliku **Zestaw.dll**.

```
Assembly a1 = Assembly.LoadFrom("Zestaw.dll");  
Assembly a2 = Assembly.Load("Zestaw");
```

Mając wczytany zestaw, w celu pobrania wszystkich typów (również zagnieżdżonych i prywatnych) w nim zdefiniowanych możemy użyć metody `GetTypes` klasy `Assembly`:

```
Assembly a = Assembly.LoadFrom("c:\\Katalog\\Przykładowy_Zestaw.dll");  
Type[] typy = a.GetTypes();
```

Po wczytaniu zestawu możemy utworzyć obiekt dowolnego typu zawartego w danym zestawie. Służy do tego klasa `System.Activator` i jego statyczna metoda `CreateInstance`:

```
Assembly a = Assembly.Load("NazwaZestawu");  
Type[] typy = a.GetTypes();  
object obj = Activator.CreateInstance(typy[0]);
```

Po utworzeniu obiektu możemy na rzecz niego wywołać metodę. Przykład wywołania metody jest umieszczony poniżej:

```
Assembly a = Assembly.Load("NazwaZestawu");  
Type typ = a.GetType("Przestrzen.Klasa");  
object obj = Activator.CreateInstance(typ);  
//Przykład wywołania metody bezparametrowej  
MethodInfo mi = typ.GetMethod("NazwaMetodyBezparametrowej");  
mi.Invoke(obj, null); //null - bez argumentów  
//Przykład wywołania metody, do której musimy przekazać argumenty  
object[] parametry = new object[2];  
parametry[0] = "Napis";  
parametry[1] = 4;  
mi = typ.GetMethod("NazwaMetodyZParametrami");  
mi.Invoke(obj, parametry);
```

Atrybuty

Wśród języków programowania istnieją również języki (tzw. *języki deklaratywne*), w których nie podajemy rozwiązania poprzez określenie sekwencji kroków, tylko przez zadeklarowanie, co chcemy uzyskać – opis tego, co nas interesuje, co ma być zrobione, ale nie jak ma być zrobione. To, jak dana rzecz ma zostać wykonana, jest rolą środowiska uruchomieniowego, kompilatora czy interpretera danego języka programowania. Idea języków deklaratywnych została przeniesiona do .NET Framework w postaci atrybutów. Atrybuty jest to mechanizm pozwalający na dodanie dodatkowej informacji opisującej poszczególne elementy kodu (np. klasę, metodę czy zdarzenie). Dodatkowe informacje opisujące nasz program i jego poszczególne elementy nazywamy *metadanymi*, czyli danymi opisujące dane. Dostęp do metadanych jest możliwy dzięki mechanizmowi refleksji, który został opisany na początku tego modułu. Zastosowanie atrybutów jest różnorodne. Pokażemy to na przykładzie atrybutów zdefiniowanych w bibliotece FCL:

- `CLSCompliantAttribute` – określa, czy dany element systemu typów (klasa, metoda, zestaw) jest zgodny ze wspólną specyfikacją języka (ang. *Common Language Specification* — CLS).
- `ObsoleteAttribute` – wskazuje elementy biblioteki, które prawdopodobnie w przyszłych jej wersjach zostaną usunięte. W przypadku gdy w swoim programie użyjemy elementu opatrzonego takim atrybutem, kompilator zgłosi ostrzeżenie.
- `ConditionalAttribute` – powoduje, że wywołanie metody jest zależne od tego, czy przy pomocy dyrektywy `#define` zdefiniowany został pewien symbol. Atrybut ten jest często wykorzystywany do metod diagnostycznych, sprawdzających pewne asercje czy informujących, że zostało osiągnięte pewne miejsce w naszym programie.
- `SerializableAttribute` – stosowany do klas, umożliwia zapisywanie stanu obiektu danej klasy do strumienia. Więcej na jego temat będzie można przeczytać w rozdziale 13 tego kursu.
- `WebMethodAttribute` – dodany do metody klasy reprezentującej usługę sieciową (ang. *XML Web service*) powoduje, że staje się ona dostępna dla klientów danej usługi.
- `DllImportAttribute` – pozwala wywołać dowolną funkcję wyeksportowaną z biblioteki DLL napisanej w kodzie niezarządzanym. Ten atrybut dodajemy do sygnatury metody, która niejako stanowi punkt wejścia do żądanej metody.

Atrybuty używamy stosując następującą składnię:

```
[nazwa_atrybutu([parametry_pozycyjne][,nazwa_parametru=wartość[,...n]])]
```

Wewnątrz nawiasów kwadratowych umieszczamy nazwę atrybutu. Z większością atrybutów związane są parametry, które umieszczamy wewnątrz nawiasów okrągłych. Mamy dwa rodzaje parametrów: *parametry pozycyjne* oraz *parametry nazwane*. Parametry pozycyjne zazwyczaj zawierają istotne dane dla danego atrybutu. Przypisane wartości parametrom pozycyjnym odbywa się na podobnej zasadzie, jak przy przekazywaniu argumentów do metody, brana jest pod uwagę kolejność argumentów. Parametry nazwane występują zawsze po parametrach pozycyjnych, są opcjonalne i składają się z par, które stanowią: nazwa parametru oraz wartość, którą należy temu parametrowi nadać, rozdzielone znakiem równości.

W celu zastosowania jakiegoś atrybutów do pewnego elementu naszego programu musimy umieścić go bezpośrednio przed danym elementem:

```
[Serializable]                                //atrybut dotyczy klasy NaszaKlasa
public class NaszaKlasa
{
    [Obsolete("Użyj innej metody")] //atrybut dotyczy metody Metoda1
    public void Metoda1()
    { }
}
```

Z jednym elementem możemy skojarzyć kilka atrybutów. W tym celu możemy umieścić poszczególne atrybuty w jednym nawiasie kwadratowym, rozdzielając je znakiem przecinka lub zapisać każdy atrybut w oddzielnych nawiasach kwadratowych, przykładowo:

```
[Serializable, Obsolete("Użyj innej!")]
public class StaraKlasa
{ }
```

lub:

```
[Serializable]
[Obsolete("Użyj innej!")]
public class StaraKlasa
{ }
```

Warto zwrócić uwagę, że nie każdy atrybut można stosować do wszystkich elementów naszego programu. Niektóre mogą być użyte tylko do klas, a inne na przykład tylko do metod.

Troszeczkę inaczej stosujemy atrybut do zestawu. Chcąc dodać metadane do zestawu, w nawiasach kwadratowych umieszczamy słowo `assembly` zakończone dwukropkiem, a dopiero później nazwę atrybutu:

```
[assembly: AssemblyCompany("Nazwa firmy")]
```

Visual Studio atrybuty dotyczące zestawu umieszcza w pliku **AssemblyInfo.cs**, który znajduje się w folderze **Properties** danego projektu.

Tworzenie własnych atrybutów

Atrybuty nie tylko są dostarczane przez bibliotekę FCL, możemy bowiem tworzyć własne atrybuty i używać ich do własnych celów.

W celu stworzenia własnego atrybutu należy:

- Napisać klasę dziedziczącą po klasie `Attribute`, zawierającą implementację zachowania dla danego atrybutu. Nazwa klasy powinna kończyć się słowem `Attribute`.
- Określić, do jakich elementów nasz atrybut może być zastosowany. Realizujemy to przez dodanie do naszej klasy atrybutu `AttributeUsageAttribute`. Do atrybutu tego przekazujemy wartość typu wyliczeniowego `System.AttributeTargets`. Wartości tego typu mogą być łączone za pomocą operatora sumy logicznej – `|`.
- Napisać dla klasy konstruktor. Parametry konstruktora będą stanowić parametry pozycyjne naszego atrybutu.
- W wypadku, gdy atrybut posiada dane opcjonalne, dostęp do nich umożliwiamy przy pomocy publicznych właściwości. Dane te będą stanowić parametry nazwane.

```
[AttributeUsage(AttributeTargets.Interface | AttributeTargets.ReturnValue |
                AttributeTargets.Parameter)]
public class MojAtrybutAttribute : System.Attribute {
    private string parametr1;
    public string Parametr1 {
        get { return parametr1; }
    }
    private string parametr2;
    public string Parametr2 {
        set { parametr2 = value; }
        get { return parametr2; }
    }
    public MojAtrybutAttribute(string parametr1) {
        this.parametr1 = parametr1;
    }
}
```

Mając powyższy atrybut możemy zastosować go do naszej klasy w następujący sposób:

```
[MojAtrybut("wartość parametru1", Parametr2="wartość parametru2")]
public interface MojInterfejs {
    void Metoda1([MojAtrybut("coś")] int x);
    [return: MojAtrybut("coś")] int Metoda2();
}
```

Zwróćmy uwagę, że końcówka `Attribute` w nazwie atrybutu została pominięta. Kompilator języka C# dodaje domyślnie przyrostek `Attribute` do nazw stosowanych atrybutów.

Wartości poszczególnych parametrów atrybutu w programie możemy wczytać w następujący sposób:

```

System.Reflection.MemberInfo memberInfo;
memberInfo = typeof(MojInterfejs);
//wartość logiczna przekazywana do metody GetCustomAttributes oznacza
//czy również przeszukiwać wszystkie elementy bazowe danego elementu w celu
//znalezienia atrybutów
object[] atrybuty = memberInfo.GetCustomAttributes(false);
foreach (Attribute atrybut in atrybuty) {
    if (atribut is MojAtrybutAttribute) {
        MojAtrybutAttribute moj = (MojAtrybutAttribute)atribut;
        Console.WriteLine("{0} {1}", moj.Parametr1,moj.Parametr2);
    }
}

```

Warto zwrócić uwagę, że atrybut `AttributeUsage` ma parametr opcjonalny `AllowMultiple`. Ustawienie wartości tego parametru na `true` powoduje, że atrybut może być stosowany wielokrotnie to tego samego elementu.

```

[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
public class MojAtrybutAttribute : System.Attribute
{ }

```

Przykładowe rozwiązanie

Korzystanie z mechanizmu wtyczek

Naszym zadaniem jest stworzenie przykładowej aplikacji rozszerzalnej przy pomocy wtyczek. Pisanie aplikacji rozpoczniemy od stworzenia interfejsu, który będzie zawierał opis funkcjonalności naszych wtyczek, czyli co chcemy, aby wtyczka robiła. Definicję interfejsu powinniśmy umieścić w oddzielnym zestawie (bibliotece DLL), ponieważ będzie z niego korzystał zarówno program główny, jak i poszczególne wtyczki. Przykładowa definicja interfejsu jest umieszczona poniżej:

```

public interface IMojPlugin {
    string Menu { get; }
    double RobCos(double x, double y);
}

```

Dodajmy następnie do tej samej biblioteki DLL, co interfejs, klasę reprezentującą atrybut. Przy pomocy tego atrybutu będziemy mogli dodać metadane (nazwisko i imię autora, opis wtyczki) do naszych wtyczek. Przykładowa definicja klasy atrybutu znajduje się poniżej:

```

[AttributeUsage(AttributeTargets.Class)]
public class MojAtrybutAttribute : Attribute {
    private string autor;
    public string Autor {
        get { return autor; }
    }

    private string opis;
    public string Opis {
        get { return opis; }
    }

    public MojAtrybutAttribute(string autor, string opis) {
        this.autor = autor;
        this.opis = opis;
    }
}

```

Utworzenie programu głównego

Program główny będzie korzystał z interfejsu `IMojPlugin`, więc nie wolno zapomnieć o dodaniu odwołania do zestawu, gdzie zdefiniowany został ten interfejs.

W programie głównym utworzymy kolekcję, w której będziemy przechowywali wszystkie załadowane wtyczki, np.:

```
private static List<IMojPlugin> zaladowaneWtyczki = new List<IMojPlugin>();
```

Potrzebna będzie również metoda, która wyświetli dostępne opcje naszego programu. Metoda powinna uwzględnić załadowane wtyczki. Przykładowa jej implementacja znajduje się poniżej:

```
private static char Menu() {
    Console.Clear();
    Console.WriteLine("\t\t\tA - Załaduj wtyczki");
    char c = 'B';
    foreach(IMojPlugin wtyczka in zaladowaneWtyczki) {
        Console.WriteLine("\t\t\t{0} - {1}", c, wtyczka.Menu);
        c++;
    }
    Console.WriteLine("\t\t\t{0} - Wyjście z programu", c);
    if (c == 'B')
        Console.WriteLine("\n\t\t\tBrak załadowanych wtyczek, "+
            + " załaduj najpierw wtyczki");
    return Console.ReadKey(true).KeyChar;
}
```

Zanim przejdziemy do metody implementującej ładowanie dostępnych wtyczek, warto zaimplementować metodę, która sprawdzi, czy dana wtyczka nie jest już załadowana. Nie chcemy ładować tej samej wtyczki kilkakrotnie:

```
private static bool CzyJuzZaladowany(Type t) {
    foreach (IMojPlugin wtyczka in zaladowaneWtyczki) {
        Type t2 = wtyczka.GetType();
        if (t == t2)
            return true;
    }
    return false;
}
```

Powinniśmy dać użytkownikowi również możliwość podjęcia decyzji, czy chce załadować daną wtyczkę. Użytkownik może podjąć wspomnianą decyzję np. na podstawie opisu danej wtyczki zawartego w atrybutach:

```
private static bool CzyZaladowac(Type t) {
    object[] atrybuty =
        t.GetCustomAttributes(typeof(MojAtrybutAttribute), false);
    if (atomytrybuty != null) {
        MojAtrybutAttribute moj = (MojAtrybutAttribute)atomytrybuty[0];
        Console.WriteLine("Czy załadować wtyczkę napisaną przez {0} " +
            "i której celem jest:\n{1}", moj.Autor, moj.Opis);
    }
    else {
        Console.WriteLine("Czy załadować wtyczkę nieznanego autora " +
            "i niewiadomego przeznaczenia?");
    }
    char c = Console.ReadKey().KeyChar;
    return c == 't' || c == 'T' ;
}
```

Teraz jesteśmy gotowi do stworzenia metody, która załaduje wtyczki z wybranego katalogu:

```

private static void ZaladujWtyczki() {
    if(!Directory.Exists("Wtyczki")) {
        Console.WriteLine("Brak katalogu Wtyczki, który służy do" +
            " przechowywania rozszerzeń tego programu!!!");
        Console.ReadKey(true);
        return;
    }
    string[] pliki = Directory.GetFiles(@"Wtyczki\", "*.dll");
    for (int i = 0; i < pliki.Length; i++) {
        Assembly zestaw = Assembly.LoadFrom(pliki[i]);
        foreach (Type typ in zestaw.GetTypes()) {
            if (typ.IsPublic && !typ.IsAbstract) {
                Type interfejsy =
                    typ.GetInterface("PluginInterfejs.IMojPlugin", true);
                if (interfejsy != null) {
                    if (!CzyJuzZaladowany(typ)) {
                        if (CzyZaladowac(typ)) {
                            IMojPlugin tmp =
                                (IMojPlugin)Activator.CreateInstance(
                                    zestaw.GetType(typ.ToString()));
                            zaladowaneWtyczki.Add(tmp);
                        }
                    }
                }
            }
        }
    }
}

```

Uzupełnijmy teraz nasz program o metodę, która wykonuje polecenie wybrane przez użytkownika. Jej implementacja może wyglądać następująco:

```

private static bool UruchomPolecenie(char c) {
    if (c == 'A')
        ZaladujWtyczki();
    int numer = c - 'B';
    if (numer == zaladowaneWtyczki.Count)
        return false; //oznacza że wybrano koniec programu

    if (numer >= 0 && numer < zaladowaneWtyczki.Count) {
        double x = 0, y = 0;
        do {
            Console.Write("Podaj pierwszą liczbę: ");
        }
        while (!double.TryParse(Console.ReadLine(), out x));
        do {
            Console.Write("Podaj drugą liczbę: ");
        }
        while (!double.TryParse(Console.ReadLine(), out y));
        Console.WriteLine("Wynikiem jest: {0}",
            zaladowaneWtyczki[numer].RobCos(x,y));
        Console.ReadKey(true);
    }
    return true;
}

```

Pozostało już tylko uzupełnić kod metody Main:

```

static void Main(string[] args) {
    char c;
    do {

```



```
        c = Menu();  
    }  
    while(UruchomPolecenie(c));  
}
```

Utworzenie własnych wtyczek

Następnym krokiem jest dodanie własnych wtyczek, czyli klas, które będą implementować interfejs IMojPlugin. Kod tych klas z oczywistych względów powinien się znaleźć w oddzielnym zestawie (bibliotece DLL). Zestaw ten powinien zawierać odwołanie do biblioteki, gdzie zdefiniowany jest interfejs IMojPlugin. Przykładowa implementacja dwóch wtyczek znajduje się poniżej:

```
[MojAtrybutAttribute("Michał Włodarczyk",  
    "Wtyczka umożliwia dodanie dwóch liczb rzeczywistych")]  
public class WtyczkaDodawanie : IMojPlugin {  
    public string Menu {  
        get { return "Dodaj dwie liczby"; }  
    }  
    public double RobCos(double x, double y) {  
        return x + y;  
    }  
}  
  
[MojAtrybutAttribute("Michał Włodarczyk",  
    "Wtyczka umożliwia odejmowanie dwóch liczb rzeczywistych")]  
public class WtyczkaOdejmowanie : IMojPlugin {  
    public string Menu {  
        get { return "Odejmij dwie liczby"; }  
    }  
    public double RobCos(double x, double y) {  
        return x - y;  
    }  
}
```

Po zbudowaniu biblioteki zawierającej implementację wtyczek wystarczy skopiować uzyskany plik DLL do katalogu **Wtyczki** znajdującego się w folderze, w którym umieszczony jest program główny.

Gotowy rozwiązany powyższy przykład znajduje się w katalogu **Demo\Modul13**.

Porady praktyczne

- Nim zastosujesz mechanizm refleksji przemyśl, czy nie można rozwiązać danego problemu w inny sposób. Refleksja jest bardzo wolna i wpływa znacząco na obniżenie wydajności Twojego programu.
- Używaj rozważnie mechanizmu wtyczek. Mechanizm ten korzysta z refleksji więc jest wolny. Stosuj go tylko wtedy, gdy planujesz, że funkcjonalność Twojej aplikacji będzie rozszerzana przez innych programistów i nie chcesz im udostępnić kodu swojego programu.
- Do nazw klas reprezentujących atrybuty dodawaj zawsze przyrostek **Attribute**, natomiast gdy stosujesz dany atrybut do dowolnego elementu kodu, omijaj ten przyrostek. Kompilator najpierw próbuje znaleźć klasę określającą typ atrybutu dodając do nazwy przyrostek **Attribute**, dopiero jeżeli jej nie znajdzie szuka klasy, której nazwa odpowiada dokładnie nazwie zastosowanego atrybutu.
- Atrybuty opisujące zestaw umieszczaj zawsze w pliku **AssemblyInfo.cs**. Plik ten nigdy nie powinien zawierać kodu Twojej aplikacji.
- Każdy atrybut powinien być umieszczony w oddzielnej linii i mieć to samo wcięcie jak element, który jest przez niego opisywany.
- Tworząc klasę reprezentującą atrybut pamiętaj, że właściwości tylko do odczytu powinny być zaimplementowane jako parametry pozycyjne. Właściwości do odczytu i zapisu utwórz natomiast jako parametry nazwane.

- W klasie reprezentującej atrybut staraj się definiować tylko jeden konstruktor. Argumenty tego konstruktora powinny służyć do inicjalizacji parametrów pozycyjnych.
- Przy pomocy atrybutu `AttributeUsageAttribute` możesz określić, do jakich elementów kodu może być użyta dana klasa reprezentująca atrybut. Przy pomocy nazwanego parametru `Inherited` atrybutu `AttributeUsageAttribute` możesz również określić, czy dany atrybut zastosowany do klasy lub składowej może być dziedziczony przez klasy pochodne lub składowe nadpisujące daną składową.
- Ze względów bezpieczeństwa staraj się używać klas zamkniętych do implementacji atrybutu.
- Przy pomocy atrybutu `CLSCompliantAttribute` możesz nakazać kompilatorowi sprawdzenie, czy Twój kod jest zgodny ze specyfikacją CLS.
- Stosuj atrybut `ConditionalAttribute` zamiast użycia dyrektywy preprocesora kompilacji warunkowej `#if` przy warunkowym wywołaniu metody, które zależy od zdefiniowania symbolu preprocesora. Pamiętaj jednak, że metoda, do której stosujesz ten atrybut jako typ zwracany, musi mieć typ `void` i nie może być to metoda implementująca funkcję jakiegoś interfejsu.
- W przypadku gdy do celów diagnostycznych używasz metody, która powinna być wywoływana tylko podczas działania aplikacji w trybie debugowania, stosuj przy jej definicji atrybut `ConditionalAttribute` z parametrem `DEBUG`.
- W przypadku gdy typ jest dostępny dla danego zestawu w czasie kompilacji, do pobierania informacji o typie, zamiast metody `Type.GetType`, używaj operatora `typeof`, gdyż jest to sposób bardziej wydajny.
- Jeżeli zamierzasz załadować zestaw tylko do celów inspekcji, np. sprawdzenia jakie typy są w nim zdefiniowane, używaj metody `Assembly.ReflectionOnlyLoad` zamiast `Assembly.Load`. Ten sposób nie pozwoli Ci wywołać kodu z załadowanego zestawu.

Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- znasz i rozumiesz pojęcie refleksji
- potrafisz uzyskać informacje o typie w czasie działania programu
- umiesz dynamicznie ładować zestaw w czasie działania programu
- potrafisz utworzyć obiekt, którego typ jest zdefiniowany w dynamicznie ładowanym zestawie i na rzecz tego obiektu wywołać metodę
- wiesz, do czego służą atrybuty
- potrafisz definiować własne atrybuty i określić do jakich elementów kodu można je stosować
- potrafisz zastosować atrybut do elementów własnego kodu
- wiesz czym się różnią parametry pozycyjne od parametrów opcjonalnych atrybutu
- potrafisz uzyskać informacje zawarte w atrybutach w czasie działania programu

Dodatkowe źródła informacji

1. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka skierowana do programistów chcących nauczyć się programować w języku C#.

2. Stephen C. Perry, *C# i .NET*, Helion, 2006

Książka w porównaniu z poprzednią skierowana do trochę bardziej zaawansowanych programistów. Opisuje C# i .NET Framework w wersji 2.0.

3. Andrew Troelsen, *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition*, Apress, 2007

Książka przeznaczona dla bardziej zaawansowanych programistów. Jej czwarte wydanie opisuje język C# 3.0 i platformę .NET 3.5.

4. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka ta nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad, jak powinniśmy pisać swoje programy.

5. Codeguru, <http://www.codeguru.pl/>

Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.

6. C Sharp Tutorial, http://www.meshplex.org/wiki/C_Sharp_Tutorial

Internetowy kurs języka C#.

7. C# Corner, <http://www.csharpcorner.com>

Portal poświęcony programowaniu w języku C#.

8. Kurs C#, cz. I, http://www.centrumxp.pl/dotNet/20,1,kategoria,Kurs_C_cz_I.aspx

Przystępny kurs języka C# w języku polskim.

Laboratorium podstawowe

Problem 1 (czas realizacji 45 min)

Firma, w której pracujesz, dysponuje kilkoma zestawami (bibliotekami), do których nie ma żadnej dokumentacji i nie bardzo wiadomo, jakie typy są w nich zdefiniowane. Twoim zadaniem jest stworzenie programu, który wczyta dany zestaw, przeanalizuje go, a następnie wygeneruje raport w postaci pliku tekstowego, który zawierać będzie informacje na temat typów zdefiniowanych w danym zestawie. Informacje, które muszą być zawarte w raporcie o każdym z typów zdefiniowanych w danym zestawie to:

- nazwa typu
- przestrzeń nazw, w jakiej zdefiniowano typ
- nazwa typu bazowego
- czy typ jest typem publicznym, zagnieżdżonym, ogólnym
- czy typ jest klasą (abstrakcyjną, zamkniętą), typem bezpośrednim, interfejsem
- lista zaimplementowanych interfejsów
- lista pól – o każdym polu muszą być zawarte następujące informacje:
 - typ pola
 - nazwa pola
 - modyfikator dostępu użyty przy definicji tego pola
 - czy pole jest polem statycznym
 - czy pole jest polem const
 - czy pole jest tylko do odczytu
- lista metod – o każdej metodzie muszą być zawarte następujące informacje:
 - nazwa metody
 - typ wartości zwracanej
 - modyfikator dostępu użyty przy definicji tej metody
 - czy metoda jest statyczna, abstrakcyjna, wirtualna, zamknięta
 - lista parametrów metody (typ i nazwa każdego parametru)
- lista konstruktorów – o każdym konstruktorze muszą być zawarte następujące informacje:
 - nazwa konstruktora
 - modyfikator dostępu użyty przy definicji konstruktora
 - czy konstruktor jest statyczny
 - lista parametrów konstruktora (typ i nazwa każdego parametru)
- lista zdarzeń – o każdym zdarzeniu muszą być zawarte następujące informacje:
 - typ delegacji
 - nazwa zdarzenia


Zadanie	Tok postępowania
1. Utwórz nowy projekt w Visual C# 2008 Express Edition typu Class Library	<ul style="list-style-type: none"> • Otwórz Visual C# 2008 Express Edition. • Z menu wybierz File -> New Project. • Z listy Visual Studio installed templates wybierz Class Library. • W polu Name wpisz InformacjeOZestawie. • Kliknij OK. • Z menu wybierz File -> Save InformacjeOZestawie. • W polu Location wybierz folder w którym będzie zapisany projekt. • Zaznacz pole wyboru Create directory for solution.

	<ul style="list-style-type: none"> W polu Solution Name wpisz Modul13. Naciśnij przycisk Save.
2. Utwórz statyczną klasę Zestaw, która będzie zawierać główną część rozwiązania problemu	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący plik Class1.cs, a następnie z menu kontekstowego wybierz Rename. Zmień nazwę pliku na Zestaw.cs. W okienku dialogowym naciśnij przycisk Tak. W pliku Zestaw.cs zaimportuj przestrzenie nazw System.Reflection i System.IO: <pre>using System.Reflection; using System.IO;</pre> Uczyń klasę Zestaw klasą statyczną: <pre>public static class Zestaw { }</pre>
3. Do klasy Zestaw dodaj metodę zapisującą do pliku informacje na temat parametrów metody	<ul style="list-style-type: none"> Do klasy Zestaw dodaj metodę statyczną zapisującą do pliku informacje na temat argumentów metody. Metoda powinna przyjmować jako argumenty: zmienną typu MethodBase reprezentującą metodę, której argumenty analizujemy oraz zmienną typu StreamWriter reprezentującą plik, do którego zapisujemy interesujące nas informacje na temat argumentów danej metody: <pre>public static void ParametryMetody(MethodBase mi, StreamWriter sw){ ParameterInfo[] parametry = mi.GetParameters(); if (parametry != null && parametry.Length > 0) { sw.WriteLine(" Do metody przesyłane są następujące parametry:"); foreach (ParameterInfo p in parametry) { sw.WriteLine(" - typ: {0}, nazwa: {1}", p.ParameterType.FullName, p.Name); } } }</pre>
4. Do klasy Zestaw dodaj metodę zapisującą do pliku informacje na temat metody	<ul style="list-style-type: none"> Do klasy Zestaw dodaj metodę statyczną zapisującą do pliku informacje na temat metody. Metoda powinna przyjmować jako argumenty: zmienną typu Type reprezentującą typ, którego metody analizujemy oraz zmienną typu StreamWriter reprezentującą plik, do którego zapisujemy interesujące nas informacje na temat metod danego typu: <pre>public static void MetodySkadowe(Type typ, StreamWriter sw) { MethodInfo[] pola = typ.GetMethods(BindingFlags.NonPublic BindingFlags.Public BindingFlags.Static BindingFlags.Instance); if (pola != null && pola.Length > 0) { sw.WriteLine(" typ zawiera następujące metody:"); foreach (MethodInfo mi in pola) { if (mi.IsPrivate) sw.Write(" - private"); if (mi.IsPublic) sw.Write(" - public"); if (mi.IsFamily) sw.Write(" - protected"); if (mi.IsAssembly) sw.Write(" - internal"); if (mi.IsFamilyOrAssembly) sw.Write(" - internal protected"); if (mi.IsStatic) sw.Write(" static "); if (mi.IsAbstract) sw.Write(" abstract "); } } }</pre>

	<pre> if (mi.IsFinal) sw.Write(" sealed "); if (mi.IsVirtual) sw.Write(" virtual "); sw.WriteLine(" {0} typ zwracany: {1}", mi.Name, mi.ReturnType); ParametryMetody(mi, sw); } } } </pre>
<p>5. Do klasy Zestaw dodaj metodę zapisującą do pliku informacje na temat konstruktorów danego typu</p>	<ul style="list-style-type: none"> Do klasy Zestaw dodaj metodę statyczną zapisującą do pliku informacje na temat konstruktorów. Metoda powinna przyjmować jako argumenty: zmienną typu Type reprezentującą typ, którego konstruktory analizujemy oraz zmienną typu StreamWriter reprezentującą plik, do którego zapisujemy interesujące nas informacje na temat konstruktorów danego typu: <pre> public static void ListaKonstruktorow(Type typ, StreamWriter sw) { ConstructorInfo[] konstr = typ.GetConstructors(BindingFlags.NonPublic BindingFlags.Public BindingFlags.Static BindingFlags.Instance); if (konstr != null && konstr.Length > 0) { sw.WriteLine(" typ zawiera następujące konstruktory:"); foreach (ConstructorInfo k in konstr) { if (k.IsPrivate) sw.Write(" - private"); if (k.IsPublic) sw.Write(" - public"); if (k.IsFamily) sw.Write(" - protected"); if (k.IsAssembly) sw.Write(" - internal"); if (k.IsFamilyOrAssembly) sw.Write(" - internal protected"); if (k.IsStatic) sw.Write(" static "); sw.WriteLine(" {0} ", k.Name); ParametryMetody(k, sw); } } } </pre>
<p>6. Do klasy Zestaw dodaj metodę zapisującą do pliku informacje na temat pól danego typu</p>	<ul style="list-style-type: none"> Do klasy Zestaw dodaj metodę statyczną zapisującą do pliku informacje na temat pól. Metoda powinna przyjmować jako argumenty: zmienną typu Type reprezentującą typ, którego pola analizujemy oraz zmienną typu StreamWriter reprezentującą plik, do którego zapisujemy interesujące nas informacje na temat pól danego typu: <pre> public static void ZmienneSkładowe(Type typ, StreamWriter sw) { FieldInfo[] pola = typ.GetFields(BindingFlags.NonPublic BindingFlags.Public BindingFlags.Static BindingFlags.Instance); if (pola != null && pola.Length>0) { sw.WriteLine(" typ zawiera następujące pola:"); foreach (FieldInfo fi in pola) { if(fi.IsPrivate) sw.Write(" - private"); if (fi.IsPublic) sw.Write(" - public"); if(fi.IsFamily) sw.Write(" - protected"); if (fi.IsAssembly) sw.Write(" - internal"); } } } </pre>

	<pre> if (fi.IsFamilyOrAssembly) sw.Write(" - internal protected"); if (fi.IsLiteral) sw.Write(" const, wartość: ", fi.GetValue(null)); if (fi.IsInitOnly) sw.Write(" readonly "); if (fi.IsStatic) sw.Write(" static "); sw.WriteLine(" {0} {1};", fi.FieldType.FullName, fi.Name); } } </pre>
7. Do klasy Zestaw dodaj metodę zapisującą do pliku informacje na temat interfejsów zaimplementowanych przez dany typ	<ul style="list-style-type: none"> Do klasy Zestaw dodaj metodę statyczną zapisującą do pliku informacje na temat interfejsów zaimplementowanych przez dany typ. Metoda powinna przyjmować jako argumenty: zmienną typu Type reprezentującą typ, który analizujemy oraz zmienną typu StreamWriter reprezentującą plik, do którego zapisujemy interesujące nas informacje na temat interfejsów zaimplementowanych przez dany typ: <pre> public static void ZaimplementowaneInterfejsy(Type typ, StreamWriter sw) { Type[] interfejsy = typ.GetInterfaces(); if (interfejsy != null && interfejsy.Length > 0) { sw.WriteLine(" typ implementuje następujące interfejsy:"); foreach (Type t in interfejsy) { sw.WriteLine(" - {0}", t.FullName); } } } </pre>
8. Do klasy Zestaw dodaj metodę zapisującą do pliku informacje na temat zdarzeń zawartych w danym typie	<ul style="list-style-type: none"> Do klasy Zestaw dodaj metodę statyczną zapisującą do pliku informacje na temat zdarzeń. Metoda powinna przyjmować jako argumenty: zmienną typu Type reprezentującą typ którego zdarzenia analizujemy oraz zmienną typu StreamWriter reprezentującą plik, do którego zapisujemy interesujące nas informacje na temat zdarzeń zawartych w danym typie: <pre> public static void Zdarzenia(Type typ, StreamWriter sw) { EventInfo[] zdarzenia = typ.GetEvents(BindingFlags.NonPublic BindingFlags.Public BindingFlags.Static BindingFlags.Instance); if (zdarzenia != null && zdarzenia.Length > 0) { sw.WriteLine(" typ zawiera następujące zdarzenia:"); foreach (EventInfo z in zdarzenia) { sw.WriteLine(" - {0} {1}", z.EventHandlerType.FullName, z.Name); } } } </pre>
9. Do klasy Zestaw dodaj metodę zapisującą do pliku informacje na temat danego typu	<ul style="list-style-type: none"> Do klasy Zestaw dodaj metodę statyczną zapisującą do pliku informacje na temat danego typu. Metoda powinna przyjmować jako argumenty: zmienną typu Type reprezentującą typ, który analizujemy oraz zmienną typu StreamWriter reprezentującą plik, do którego zapisujemy interesujące nas informacje na temat danego typu: <pre> public static void InformacjeOTypie(Type typ, StreamWriter sw) { sw.WriteLine(" - {0}", typ.Name); sw.WriteLine(" zdefiniowany jest w przestrzeni nazw: {0}", typ.Namespace); sw.WriteLine(" typ bazowy: {0}", typ.BaseType.FullName); sw.WriteLine(" czy typ jest typem publicznym: {0}", typ.IsPublic); sw.WriteLine(" czy typ jest typem zagnieżdżonym: {0}", </pre>

	<pre> typ.IsNested); sw.WriteLine(" czy typ jest typem ogólnym: {0}", typ.IsGenericType); if (typ.IsClass) { sw.Write(" typ jest klasą "); if(typ.IsAbstract) sw.Write(" abstrakcyjną "); if (typ.IsSealed) sw.Write(" zamkniętą "); sw.WriteLine(); } if (typ.IsInterface) { sw.WriteLine(" typ jest interfejsem"); } if (typ.IsValueType) { sw.WriteLine(" typ jest typem bezpośrednim"); } ZaimplementowaneInterfejsy(typ, sw); ListaKonstruktorow(typ, sw); ZmienneSkladowe(typ, sw); MetodySkladowe(typ, sw); Zdarzenia(typ, sw); sw.WriteLine("-----"); } </pre>
10. Do klasy Zestaw dodaj metodę zapisującą do pliku raport o danym zestawie	<ul style="list-style-type: none"> Do klasy Zestaw dodaj metodę statyczną zapisującą do pliku raport o danym zestawie. Metoda powinna przyjmować jako argumenty: zmienną typu Assembly reprezentującą zestaw, który analizujemy oraz zmienną typu StreamWriter reprezentującą plik, do którego zapisujemy interesujące nas informacje na temat danego zestawu: <pre> public static void SporzadzRaport(Assembly asm, StreamWriter sw) { sw.WriteLine("****Informacje o zestawie {0}****\n", asm.FullName); sw.WriteLine("Powyższy zestaw zawiera następujące typy:"); Type[] typy = asm.GetTypes(); foreach (Type typ in typy) { InformacjeOTypie(typ, sw); } } </pre>
11. Do bieżącego rozwiązania dodaj nowy projekt	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący rozwiązanie, a następnie z menu kontekstowego wybierz Add -> New Project. W oknie dialogowym Add New Project z listy Visual Studio installed templates wybierz Console Application. W polu Name wpisz EksploratorTypow. Kliknij OK.
12. Zaznacz projekt EksploratorTypow jako projekt startowy	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element reprezentujący projekt EksploratorTypow, a następnie z menu kontekstowego wybierz Set as StartUp Project.
13. Do projektu EksploratorTypow dodaj odwołanie do biblioteki InformacjeOZestawie	<ul style="list-style-type: none"> W okienku Solution Explorer zaznacz prawym klawiszem myszy element o nazwie References w drzewie projektu EksploratorTypow, a następnie z menu kontekstowego wybierz Add Reference. W oknie dialogowym Add Reference przejdź do zakładki Projects, zaznacz projekt InformacjeOZestawie, a następnie kliknij przycisk OK.
14. Zaimplementuj metodę Main	<ul style="list-style-type: none"> Przejdź do pliku Program.cs. Na górze pliku Program.cs zaimportuj przestrzenie nazw System.Reflection, System.IO i InformacjeOZestawie:

	<pre>using System.Reflection; using System.IO; using InformacjeOZestawie;</pre> <ul style="list-style-type: none"> Do metody Main dodaj kod, który pobierze z linii poleceń nazwę zestawu do analizy oraz nazwę pliku raportu, a następnie wygeneruje raport: <pre>static void Main(string[] args) { if(args.Length < 2) { Console.WriteLine("Program należy wywołać:"); Console.WriteLine("EksploratorTypow <nazwa zestawu do badania>" +" <nazwa pliku z raportem>"); Console.ReadKey(); return; } if(!File.Exists(args[0])) { Console.WriteLine("Podany zestaw do badania nie istnieje!!!"); Console.ReadKey(); return ; } StreamWriter sw = null; try { Assembly asembly = Assembly.LoadFrom(args[0]); sw = new StreamWriter(args[1]); Zestaw.SporzadzRaport(assembly, sw); Console.WriteLine("Raport został sporządzony!!!"); } catch (IOException ex) { Console.WriteLine("Uwaga wyjątek!!!\n{0}", ex.Message); } finally { if (sw != null) sw.Close(); } Console.ReadKey(); }</pre>
15. Skompiluj i uruchom program	<ul style="list-style-type: none"> Z menu Build wybierz Build Solution. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program. W celu uruchomienia programu z menu Debug wybierz Start Debugging.
16. Przetestuj działanie programu	<ul style="list-style-type: none"> Do katalogu Bin\Debug, który znajduje się w katalogu bieżącego projektu, skopiuj plik Osoby.dll, który znajduje się w katalogu Kurs\Lab\Start\Modul12, gdzie Kurs jest katalogiem, w którym zainstalowano pliki kursu. Z menu Project wybierz EksploratorTypow Properties.  Zwróć uwagę, aby przed wykonaniem powyższego polecenia w okienku Solution Explorer był zaznaczony projekt EksploratorTypow. W okienku właściwości projektu EksploratorTypow wybierz zakładkę Debug. W polu tekstowym Command line arguments wpisz Osoby.dll Raport.txt. Uruchom program. Z menu Debug wybierz Start Debugging. Przejrzy plik testowy Raport.txt, który powinien znajdować się w katalogu Bin\Debug, umieszczonym w katalogu bieżącego projektu.

Laboratorium rozszerzone

Zadanie 1 (czas realizacji 90 min)

Firma, w której pracujesz, dostała zlecenie napisania programu graficznego od pewnej agencji reklamowej. Program ma stosować różnorodne filtry graficzne do obrazków wykorzystywanych przez agencję reklamową. Ponieważ nowe filtry będą się pojawiać wraz z upływem czasu, do implementacji programu zdecydowano się użyć mechanizmu wtyczek. Twoim zadaniem jest napisanie programu głównego, opracowanie interfejsu stosowanego przez wtyczki oraz napisanie kilku wtyczek, wymaganych już na wstępie przez agencję reklamową. Wtyczka powinna również zawierać krótki opis do czego służy oraz nazwisko i imię jej autora. Filtry które musisz zaimplementować to (w poniższym opisie litery R, G, B oznaczają natężenie koloru odpowiednio: czerwonego, zielonego i niebieskiego. Pamiętaj, że natężenie może mieć wartość od zera do 255):

- Rozjaśnienie lub przyciemnienie: polega na zwiększeniu lub zmniejszeniu wartości składowych R, G i B każdego punktu tworzącego bitmapę. Każdą ze składowych zwiększamy lub zmniejszamy o tę samą wartość (pewien współczynnik).
- Zmiana kontrastu: pseudokod zostanie podany dla składowej czerwonej. Pozostałe składowe oblicza się w analogiczny sposób:

```
if(wspolczynnik > 0) {                //zwiększamy kontrast
    if(R<wspolczynnik)
        R = 0;
    else
        if(R>255-wspolczynnik)
            R = 255;
        else
            R = 255/(255-2*wspolczynnik) * (R-wspolczynnik);
}
else
    if (wspolczynnik<0) {              //zmniejszamy kontrast
        R = -wspolczynnik+(255+2*wspolczynnik)/255 * R;
    }
```

- Negatyw: każde natężenie koloru będziemy obliczać następująco:

```
R = ~R;
G = ~G;
B = ~B;
```

- Wyzerowanie wybranych składowych koloru – np. wyzerowanie koloru czerwonego:

```
R = 0;
```

- Zmiana balansu kolorów: każdą ze składowych koloru zmieniamy o inny współczynnik:

```
R += wspolczynnikR;
G += wspolczynnikG;
B += wspolczynnikB;
```

- Zmiana zdjęcia kolorowego na zdjęcie w odcieniach szarości:

```
R = B = G = (R + G + B) / 3;
```

lub z uwzględnieniem czułości oka na poszczególne barwy składowe

```
R = B = G = 0,3 * R + 0,59 * G + 0,11 * B
```

Do wczytania obrazka możesz użyć klasy `System.Drawing.Bitmap`, zdefiniowanej we współdzielonym zestawie `System.Drawing` w następujący sposób:

```
Bitmap bmp = new Bitmap("nazwa pliku (gif, jpg, bmp itp.)");
```

Natomiast zmodyfikowany obrazek możesz zapisać przy pomocy metody `Save` klasy `Bitmap`. Do pobrania koloru wybranego punktu oraz ustawienie koloru wybranego punktu możesz użyć odpowiednio metod `Bitmap.GetPixel` i `Bitmap.SetPixel`. Szerokość i wysokość w pikselach wczytanego obrazu można określić odpowiednio przy pomocy właściwości `Width` oraz `Height` klasy `Bitmap`. Do reprezentacji koloru możesz zastosować strukturę `Color`. Nowy kolor możesz utworzyć następująco:

```
Color c = Color.FromArgb(R,G,B);
```

Natężenie poszczególnych składowych (R, G, B) możesz uzyskać w następujący sposób:

```
int czerwony = c.R;  
int zielony = c.G;  
int niebieski = c.B;
```


ITA-105 Programowanie obiektowe

Michał Włodarczyk

Moduł 14

Wersja 2

Serializacja

Spis treści

Serializacja	1
Informacje o module.....	2
Przygotowanie teoretyczne.....	3
Przykładowy problem	3
Podstawy teoretyczne.....	3
Przykładowe rozwiązanie	9
Porady praktyczne	14
Uwagi dla studenta	15
Dodatkowe źródła informacji	15
Laboratorium podstawowe	15
Problem 1 (czas realizacji 35 min)	15
Problem 2(czas realizacji 10 min)	21
Laboratorium rozszerzone	26
Zadanie (czas realizacji 90 min)	26

Informacje o module

Opis modułu

W tym module zapoznasz się z pojęciem serializacji. Poznasz rodzaje serializacji wspierane przez .NET Framework oraz nauczysz się, jak tworzyć klasy serializowalne i ich używać. Zobaczysz również, jak można dostosować proces serializacji, aby spełniał Twoje oczekiwania. W module opisana jest serializacja binarna oraz serializacja XML.

Cel modułu

Celem modułu jest przedstawienie możliwości wykorzystania pewnych cech środowiska .NET Framework w celu zapisania stanu obiektu.

Uzyskane kompetencje

Po zrealizowaniu modułu będziesz:

- wiedział, co to jest serializacja
- znał rodzaje serializacji wspierane przez .NET Framework
- potrafił tworzyć klasy serializowalne
- potrafił korzystać z klas serializowalnych
- potrafił dostosowywać serializację do własnych potrzeb

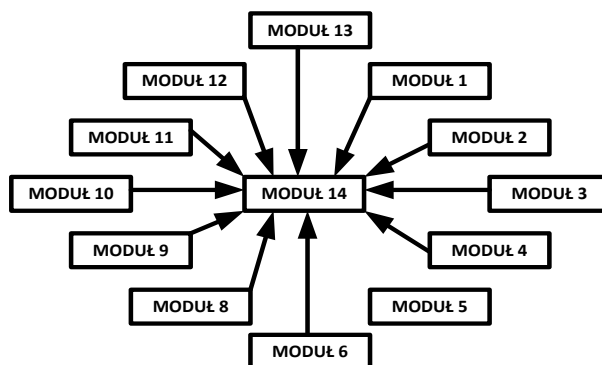
Wymagania wstępne

Przed przystąpieniem do pracy z tym modulem powinieneś:

- potrafić korzystać ze strumieni w języku C#
- potrafić stosować atrybuty
- znać pojęcie interfejsu
- wiedzieć, co to jest zdarzenie
- znać podstawy języka XML

Mapa zależności modułu

Zgodnie z mapą zależności przedstawioną na rys. 1, przed przystąpieniem do realizacji tego modułu należy zapoznać się z materiałem zawartym w modułach Pojęcie klasy, „Właściwości i indeksatory”, „Składowe statyczne”, „Dziedziczenie”, „Polimorfizm – funkcje wirtualne”, „Klasy abstrakcyjne i interfejsy”, „Typy ogólne”, „Implementacja przykładowych interfejsów”, „Delegacje i zdarzenia” oraz „Refleksja i atrybuty”,.



Rys. 21 Mapa zależności modułu

Przygotowanie teoretyczne

Przykładowy problem

Jesteś programistą w firmie komputerowej. Firma ma do wykonania pewien system informatyczny. Jesteś odpowiedzialny za implementację jednej z klas wchodzących w jego skład. Oprócz wymagań dotyczących pól, właściwości i metod, dodatkowym żądaniem jest, aby obiekty tej klasy były łatwo zapisywalne na dysku oraz aby można je było przysyłać między różnymi aplikacjami. Musisz też zwrócić uwagę, że system prawdopodobnie będzie rozwijany przez dłuższy okres i będą konieczne zmiany w implementacji Twojej klasy przez dodanie lub usunięcie pewnych pól klasy. Trzeba zapewnić, żeby aplikacje oparte na różnych wersjach Twojej klasy współpracowały ze sobą. Aplikacja w nowszej wersji powinna mieć możliwość wczytania obiektów zapisanych lub przesłanych przez aplikację w wersji starszej, a aplikacja oparta na starszej wersji klasy musi „rozumieć” nowe wersje obiektów.

Kolejnym problemem, który musisz rozwiązać jest to, że obiekty Twojej klasy są polami innych klas, również klas kolekcji, które mogą być zapisywane na dysku. W swojej implementacji musisz zapewnić, że obiekt będzie raz umieszczony w strumieniu bez względu na to, ile razy odwołują się do niego elementy kolekcji. Musi być też zapewniony poprawny sposób wyjmowania obiektów ze strumienia, żeby nie zgubić wzajemnych zależności.

Następnym wyzwaniem jest to, że nie wszystkie pola mogą być bezpośrednio umieszczone w strumieniu. Niektóre mogą zostać pominięte, a niektóre przed wystąpieniem do strumienia wstępnie przetworzone, np. zaszyfrowane. Oczywiście w momencie wyjmowania ze strumienia musimy mieć możliwość deszyfrowania pól oraz nadania wartości nieprzesłanym polom.

Zastanowiłeś się chwilę nad sposobem implementacji swojej klasy. Zrozumiałeś, że trzeba opracować jednolity mechanizm umieszczania obiektów w strumieniu. Mechanizm ten powinien być wspólny dla wszystkich klas tworzonych w firmie, których obiekty mogą być umieszczone w strumieniu i powinien uwzględniać kontrolę wersji obiektu, kontrolę redundancji oraz możliwość określenia, które pola mają być pominięte i ich wstępne przetworzenie przed umieszczeniem w strumieniu, jak i zaraz po wyjęciu z niego. Całe szczęście nie musisz tworzyć tego mechanizmu od podstaw. Jest on już zaimplementowany w .NET Framework i nazywa się serializacja.

Podstawy teoretyczne

Serializacja (ang. *serialization*) jest to proces konwersji stanu obiektu do postaci, która może być zachowana lub przesłana. Proces odwrotny, czyli odzyskanie stanu obiektu ze strumienia, nazywamy *deserializacją* (ang. *deserialization*). Serializację jest łatwa w zastosowaniu, o czym będzie można się przekonać za chwilę, jest to również standardowy sposób zapisu stanu obiektu – w .NET Framework jest on używana przy przesyłaniu obiektów przez wartość z jednej domeny aplikacji do drugiej. Serializację stosują podstawowe technologie, które często się używa, tworząc oprogramowanie na platformę .NET, np.:

- .NET remoting
- XML Web Services
- Windows Communication Foundation

Na platformie .NET wyróżniamy trzy rodzaje serializacji:

- serializacja XML (ang. *XML Serialization*).
- serializacja do protokołu SOAP (ang. *SOAP Serialization*).
- serializacja binarna (ang. *Binary Serialization*).

Serializacja XML

Serializacja XML zapisuje (serializuje) tylko wartości publicznych pól i wartości publicznych właściwości do odczytu i zapisu obiektu danej klasy. Klasa, której obiekt serializujemy, musi być publiczna oraz posiadać publiczny bezparametrowy konstruktor. Niespełnienie tego warunku powoduje zgłoszenie wyjątku `InvalidOperationException`. Serializacja XML nie zachowuje informacji o typie serializowanego obiektu, więc nie ma gwarancji, że po deserializacji otrzymamy obiekt, którego typ będzie identyczny z typem oryginału. Ostatnia właściwość może wydawać się wadą, ale bardzo się przydaje, gdy próbujemy wymieniać dane z aplikacją, która nie jest napisana na platformę .NET i nie używa wspólnego systemu typów (ang. *Common Type System*). Dzięki językowi XML wymiana informacji jest możliwa.

Do serializacji XML używamy klasy `XmlSerializer`. Jej definicja znajduje się w podzespole `System.Xml` w przestrzeni nazw `System.Xml.Serialization`. Dwie najważniejsze metody tej klasy to `Serialize` i `Deserialize`. Przykład użycia klasy `XmlSerializer` do serializacji obiektu klasy `Adres` można zobaczyć poniżej. Oczywiście strumień plikowy `FileStream` może być zastąpiony strumieniem innego typu:

```
XmlSerializer serializer = new XmlSerializer(typeof(Adres));
using(FileStream fs = new FileStream("Adres.xml", FileMode.Create);) {
    serializer.Serialize(fs, zmiennaTypuAdres);
}
```

Zakładając, że klasa `Adres` jest zdefiniowana w następujący sposób:

```
public class Adres{
    public string Miejscowosc { set; get; }
    public string Ulica { set; get; }
    public int NumerDomu { set; get; }
    public int? NumerMieszkania { set; get; }
}
```

W wyniku działania powyższego kodu otrzymamy następujący plik **Adres.xml**:

```
<?xml version="1.0"?>
<Adres xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Miejscowosc>Piotrków Tryb.</Miejscowosc>
  <Ulica>Kwiatowa</Ulica>
  <NumerDomu>341</NumerDomu>
  <NumerMieszkania xsi:nil="true" />
</Adres>
```

Analizując powyższy przykład możemy zauważyć, że element główny dokumentu XML posiada nazwę klasy serializowanej, natomiast publiczne właściwości do odczytu i zapisu (dotyczy to również publicznych pól) zostały zapisane jako podelementy o nazwach odpowiadających nazwom tych właściwości (ewentualnie pól). Jeżeli typ publicznej właściwości lub pola nie byłby typem prostym, tylko obiektem jakiejś klasy lub struktury, to wewnątrz elementu odpowiadającego klasie zawierającej tę właściwość, jest tworzony element o nazwie danej właściwości. Element ten oczywiście będzie zawierał podelementy reprezentujące publiczne właściwości i pola jego klasy.

Nie zawsze domyślny sposób działania serializacji musi być zgodny z naszymi oczekiwaniami. W takim przypadku możemy skorzystać z atrybutów serializacji, za pomocą których można w prosty sposób dostosować do własnych potrzeb funkcjonowanie tego procesu. Poniżej przedstawiono kilka atrybutów, stosowanych chyba najczęściej:

- `XmlElementAttribute` – kontroluje właściwości elementu XML takie jak np. nazwa elementu dla danej właściwości lub pola:

- ```
[XmlElement(ElementName = "NazwaUlicy")]
public string Ulica;
```
- `XmlIgnoreAttribute` – powoduje, że dane pole lub właściwość jest ignorowane podczas serializacji (nie zostanie umieszczone w docelowym strumieniu):
 

```
[XmlIgnore]
public string poleNieserializowane;
```
  - `XmlAttributeAttribute` – powoduje, że dana właściwość lub pole będzie osadzone w postaci atrybutu, a nie podelementu. Podobnie jak w przypadku `XmlElementAttribute`, możemy określić nazwę atrybutu:
 

```
[XmlAttribute("NazwaAtrybutu")]
public string NazwaPola
```
  - `XmlTextAttribute` – powoduje, że wartość danego pola lub właściwości będzie przekształcona na tekst – nie będzie dla niej tworzony element. Ten atrybut w klasie może wystąpić tylko raz, w przeciwnym wypadku zostanie zgłoszony wyjątek. Może być stosowany, gdy typ pola lub właściwości jest typem podstawowym lub typem wyliczeniowym. Typ ten może być również tablicą napisów (stringów), tablicą elementów typu `object`, klasą `XmlNode` lub tablicą elementów typu `XmlNode`.
 

```
[XmlText]
public string NazwaPola;
```
  - `XmlRootAttribute` – w odróżnieniu od poprzednich atrybutów może być stosowany do klasy, struktury, typu wyliczeniowego lub interfejsu. Kontroluje, jak ma być utworzony element główny dokumentu XML:
 

```
[XmlRoot(Namespace = "www.kurs.microsoft.pl/programowanieobektowe",
 ElementName = "KlasaOsoba")]
public class Osoba {...}
```

W celu odzyskania obiektu, czyli dokonania jego deserializacji, używamy metody `Deserialize` klasy `XmlSerializer`. Zwróćmy uwagę, że metoda `Deserialize` zwraca typ `object`, należy więc wynik wywołania tej metody rzutować na odpowiedni typ. Przykład deserializacji jest umieszczony poniżej:

```
XmlSerializer serializer = new XmlSerializer(typeof(Adres));
FileStream fs = null;
try{
 fs = new FileStream("NazwaPliku.xml", FileMode.Open);
 Adres odzyskany = (Adres)serializer.Deserialize(fs);
}
finally{
 if(fs != null)
 fs.Close();
}
```

### ***Serializacja do protokołu SOAP***

Ten rodzaj serializacji wykorzystuje klasę `SoapFormatter` zamiast `XmlSerializer` i zapisuje dane w wiadomości SOAP (ang. *Simple Object Access Protocol*) – również do postaci kodu języka XML, ale zgodnego ze standardami protokołu SOAP. Serializację do protokołu SOAP stosuje się w bardzo podobny sposób, jak serializację binarną, która jest opisana w następnym rozdziale. Niestety, kurs ten nie opisuje serializacji do protokołu SOAP. Jeżeli jesteś zainteresowany serializacją do protokołu SOAP, skorzystaj z rozdziału „Dodatkowe źródła informacji” w tym module.

### **Serializacja binarna**

Serializacja binarna w odróżnieniu od serializacji XML zapisuje wszystkie pola serializowanego obiektu, również składowe prywatne. Dodatkowo zapisywana jest pełna informacja o typie serializowanego obiektu, jak i jego pól. Informacja o typie zawiera w pełni kwalifikowaną nazwę typu oraz nazwę podzespołu (nazwę zestawu, kulturę, wersję oraz skrót klucza publicznego), gdzie dany typ jest zaimplementowany. Dzięki temu podczas deserializacji otrzymujemy obiekt, którego typ jest zgodny z typem serializowanego obiektu. W dalszej części rozdziału **typem serializowalnym** będziemy nazywać typ, którego obiekty (zmienne) mogą być serializowane binarnie.

Klasy, której obiekty chcemy serializować binarnie, tworzymy przez dodanie do jej deklaracji atrybutu `SerializableAttribute`:

```
[Serializable]
class NazwaKlasy{...}
```

Ważne jest, żeby klasa bazowa danej klasy także była serializowalna, czyli opatrzona atrybutem `SerializableAttribute`. Warunek ten dotyczy również typów pól (tych, które będą zapisywane). W przypadku gdy klasa bazowa lub któryś typ pola naszej klasy nie będzie opatrzony atrybutem `SerializableAttribute`, wtedy podczas próby serializacji obiektu naszej klasy zostanie rzucony wyjątek `SerializationException`. Całe szczęście, że większość typów w bibliotece FCL jest oznaczonych tym atrybutem. Również typy wbudowane, w tym typ `object`, są serializowalne. Podobnie nie będziemy mieli problemów z tablicami obiektów klas serializowalnych.

Do serializacji binarnej używamy klasy `BinaryFormatter` z przestrzeni nazw `System.Runtime.Serialization.Formatters.Binary`. Implementacja jej znajduje się w bibliotece `mscorlib.dll`. Podobnie jak klasa `SoapFormatter`, klasa `BinaryFormatter` implementuje interfejs `IFormatter`, który posiada dwie metody: `Serialize` i `Deserialize`. Poniższy przykład demonstruje użycie metody `Serialize` do zapisu stanu obiektu do pliku. Oczywiście strumień plikowy `FileStream` może być zastąpiony strumieniem innego typu.

```
BinaryFormatter formatter = new BinaryFormatter();
FileStream fs = null;
try{
 fs = new FileStream("nazwaPliku", FileMode.Create);
 formatter.Serialize(fs, nazwaZmiennej);
}
finally{
 if(fs != null)
 fs.Close();
}
```

Odzyskać obiekt ze strumienia przy pomocy metody `Deserialize` możemy w następujący sposób:

```
BinaryFormatter formatter = new BinaryFormatter();
FileStream fs = null;
TypSerializowany nazwaZmiennej;
try{
 fs = new FileStream("nazwaPliku", FileMode.Open);
 nazwaZmiennej = (TypSerializowany)formatter.Deserialize(fs);
}
finally{
 if (fs != null)
 fs.Close();
}
```

Rozważmy teraz definicję następującej klasy:

```
[Serializable]
class Osoba{
```

```
 public string Imie, Nazwisko;
 public Osoba Partner;
}
```

oraz następujący kod:

```
Osoba maz = new Osoba { Imie = "Jan", Nazwisko = "Kowalski" };
Osoba zona = new Osoba { Imie = "Janina", Nazwisko = "Kowalska" };
maz.Partner = zona;
zona.Partner = maz;
```

Powinno nas zastanowić, czy można w bezpieczny sposób serializować tak zdefiniowany obiekt o nazwie `maz`, ewentualnie `zona`, czyli czy można serializować obiekty, których pola odwołują się do siebie nawzajem, tworząc niejako pętle. Łatwo domyślić się, że odpowiedź brzmi tak. Zawdzięczamy to temu, że podczas serializacji środowisko uruchomieniowe (CLR) tworzy wykres serializowanego obiektu (ang. *Object Graphs*). Podczas tworzenia wykresu obiektu każdemu elementowi składowemu, który ma być serializowany, przyporządkowywany jest unikalny numer, który w jednoznaczny sposób identyfikuje dany obiekt. Troszeczkę upraszczając możemy przyjąć, że podczas serializacji zamiast obiektu, który jest polem serializowanego obiektu, zapisywany jest tylko jego numer, sam obiekt jest zaś zapisany oddzielnie. Dzięki temu mamy pewność, że każdy obiekt zostanie zapisany tylko raz i podczas deserializacji zostaną odtworzone poprawne relacje między poszczególnymi obiektami.

### Dostosowanie serializacji do potrzeb użytkownika

Podobnie jak w przypadku serializacji XML, serializację binarną możemy również dostosować do własnych potrzeb. Klasa może zawierać pola, których wartości nie chcemy zachowywać (przesyłać), np. z przyczyn bezpieczeństwa. Realizujemy to poprzedzając takie pole atrybutem `NonSerializedAttribute`:

```
[NonSerialized]
private typPola nazwaPolaNieserializowanego;
```

Podczas deserializacji polu nieserializowanemu zostanie przypisana wartość domyślna dla danego typu (dla typów numerycznych wartość zero, dla typów referencyjnych `null` itp.). Chcąc nadać inną wartość takiemu polu możemy zaimplementować w danej klasie interfejs `System.Runtime.Serialization.IDeserializationCallback`. Interfejs ten posiada metodę `OnDeserialization`, która zostaje wywołana po „odzyskaniu” obiektu:

```
[Serializable]
class TwojaKlasa: IDeserializationCallback {
 [NonSerialized]
 private typPola nazwaPolaNieserializowanego;
 public void OnDeserialization(object sender){
 nazwaPolaNieserializowanego = ŻądanaWartość;
 }
}
```

Bardziej kontrolować proces serializacji możemy przez implementację interfejsu `ISerializable` z przestrzeni nazw `System.Runtime.Serialization`. Implementacja tego interfejsu polega na dodaniu do naszej klasy metody `GetObjectData` oraz specjalnego konstruktora, który jest wywoływany podczas procesu deserializacji. Przykład implementacji interfejsu `ISerializable` jest zamieszczony poniżej:

```
[Serializable]
public class MojaKlasa: ISerializable {
 public int Pole1;
 public double Pole2;
 public string Pole3;
```

```
public MojaKlasa(){ }

protected MojaKlasa(SerializationInfo info, StreamingContext context){
 Pole1 = info.GetInt32("Klucz1") - 1;
 Pole3 = info.GetString("Klucz3");
 Pole2 = 3.33;
}

public void GetObjectData(SerializationInfo info,
 StreamingContext context){
 info.AddValue("Klucz1", Pole1+1);
 info.AddValue("Klucz3", Pole3);
}
}
```

Jak widać w powyższym przykładzie, w metodzie `GetObjectData` podajemy, co ma być zapisane jako stan naszego obiektu. Przed zapisaniem wartość możemy zmodyfikować (patrz `Pole1`). W konstruktorze odzyskujemy zapisane wartości i na ich podstawie możemy odtworzyć stan obiektu.

W wersji 2.0 .NET Framework wprowadzono pewne atrybuty, które w znaczny sposób ułatwiają kontrolę procesu serializacji. Należą do nich:

- `OnDeserializedAttribute` – stosowany jest do metody, która będzie wywołana tuż po deserializacji obiektu danej klasy. Zastępuje niejako implementację interfejsu `IDeserializationCallback`.
- `OnDeserializingAttribute` – stosowany jest do metody, która będzie wywołana na początku procesu deserializacji.
- `OnSerializedAttribute` – stosowany jest do metody, która będzie wywołana tuż po serializacji obiektu danej klasy.
- `OnSerializingAttribute` – stosowany jest do metody, która będzie wywołana na początku procesu serializacji.

Metody poprzedzone powyższymi atrybutami, w odróżnieniu od implementacji interfejsu `ISerializable`, nie korzystają bezpośrednio ze strumienia do lub z którego zapisujemy lub odczytujemy obiekt danej klasy, ale pozwalają nam zmodyfikować pola naszego obiektu na początku lub końcu procesu serializacji lub deserializacji. Metoda, którą chcemy poprzedzić jednym z powyższych atrybutów, musi zwracać typ `void` i przyjmować jako argument zmienną typu `StreamingContext`. Przykład użycia powyższych atrybutów jest umieszczony poniżej:

```
[Serializable()]
public class MojaKlasa{
 public string Pole1;
 public string Pole2;
 [NonSerialized()]
 public string Pole3;
 public string Pole4;

 public MojaKlasa() { }

 [OnSerializing()]
 internal void OnSerializingMethod(StreamingContext context){
 Pole2 = "Szyfrujemy przed serializacja.";
 }

 [OnSerialized()]
 internal void OnSerializedMethod(StreamingContext context){
 Pole2= "Przywracamy wartość po serializacji.";
 }
}
```

```
[OnDeserializing()]
internal void OnDeserializingMethod(StreamingContext context){
 Pole3 = "Tego pola i tak nie ma w strumieniu";
}

[OnDeserialized()]
internal void OnDeserializedMethod(StreamingContext context){
 Pole4 = "Nadpisujemy wartosc odczytaną ze strumienia.";
}
}
```

## Przykładowe rozwiązanie

### *Serializacja binarna*

Naszym zadaniem jest utworzenie klasy *Osoba* składającej się z następujących pól i właściwości:

- publicznych właściwości *Imie* i *Nazwisko* typu *string*
- prywatnego pola *rokRodzenia* typu *int*
- prywatnego pola *haslo* typu *string*

Klasa musi być serializowalna binarnie. Pole reprezentujące rok urodzenia przed zapisaniem do strumienia powinno być zaszyfrowane przy pomocy funkcji XOR, oczywiście przy deserilizacji pole powinno być prawidłowo deszyfrowane. Pole reprezentujące hasło nie powinno być zapisywane. Podczas odzyskiwania ze strumienia hasło powinno być utworzone przez konkatencję imienia i nazwiska.

### Implementacja serializacji binarnej w klasie *Osoba*

Przykładowa początkowa implementację klasy *Osoba* może wyglądać następująco:

```
public class Osoba{
 public string Imie { set; get; }
 public string Nazwisko { set; get; }

 private int rokUrodzenia;
 public int Wiek{
 get { return DateTime.Now.Year - rokUrodzenia; }
 }

 private string haslo;
 public bool SprawdzHaslo(string hasloTest){
 if (haslo == hasloTest)
 return true;
 return false;
 }

 public bool UstawHaslo(string stare, string nowe){
 if (SprawdzHaslo(stare)){
 haslo = nowe;
 return true;
 }
 return false;
 }

 public Osoba(int rokUrodzenia,string haslo){
 this.rokUrodzenia = rokUrodzenia;
 this.haslo = haslo;
 }
}
```

Zaznaczmy, że klasa `Osoba` jest serializowalna przez dodanie do niej atrybutu `SerializableAttribute`:

```
[Serializable]
public class Osoba {...}
```

Zaznaczmy, że pole `haslo` nie powinno być serializowane przez dodanie do niego atrybutu `NonSerializedAttribute`:

```
[NonSerialized]
private string haslo;
```

Do klasy `Osoba` dodajmy pole prywatne `Maska`, które będzie służyć jako zmienna pomocnicza służąca do szyfrowania i deszyfrowania pola `rokUrodzenia`. Dodajmy również metodę prywatną `PrzedSerializacja`. Metoda ta będzie zwracać typ `void` i przyjmować jako argument zmienną typu `StreamingContext`, aby można było dodać do niej odpowiednie atrybuty związane z serializacją. Pamiętajmy, że klasa `StreamingContext` znajduje się w przestrzeni nazw `System.Runtime.Serialization`, więc należy ją zaimportować lub stosować w pełni kwalifikowane nazwy przy korzystaniu z typów w niej zdefiniowanych. Celem metody jest zaszyfrowanie pola `rokUrodzenia` przed serializacją. Do metody tej dodajmy atrybut `OnSerializingAttribute`, aby była ona automatycznie wywoływana na początku procesu serializacji:

```
[OnSerializing]
private void PrzedSerializacja(StreamingContext context){
 rokUrodzenia ^= Maska;
}
```

Do klasy `Osoba` dodajmy metodę prywatną `PoSerializacji`. Celem metody `PoSerializacji` jest przywrócenie prawidłowej wartości polu `rokUrodzenia`. Do metody tej dodajmy atrybut `OnSerializedAttribute`, aby była ona automatycznie wywołana na koniec procesu serializacji. Oczywiście aby móc dodać ten atrybut metoda musi zwracać typ `void` i przyjmować jako argument zmienną typu `StreamingContext`:

```
[OnSerialized]
private void PoSerializacji(StreamingContext context){
 rokUrodzenia ^= Maska;
}
```

Do klasy `Osoba` dodajmy metodę prywatną `PoDeserializacji`. Metoda ta, podobnie jak poprzednie, będzie zwracać typ `void` i przyjmować jako argument zmienną typu `StreamingContext`. Celem metody `PoDeserializacji` jest przywrócenie prawidłowej wartości polu `rokUrodzenia` oraz nadanie odpowiedniej wartości polu `haslo`. Do metody tej dodajmy atrybut `OnDeserialized`, aby była ona automatycznie wywołana na koniec procesu deserializacji:

```
[OnDeserialized]
private void PoDeserializacji(StreamingContext context){
 rokUrodzenia ^= Maska;
 haslo = Imie + Nazwisko;
}
```

## Utworzenie programu testowego

### *Test serializacji binarnej*

W metodzie `Main` utworzymy tablicę obiektów klasy `Osoba` i inicjalizujemy te obiekty znanymi sobie wartościami. Utworzymy następnie obiekt klasy `BinaryFormatter` oraz zmienną klasy `FileStream`, która będzie reprezentować strumień, do którego prześlemy utworzone obiekty klasy `Osoba`. W bloku `try` skojarzymy zmienną klasy `FileStream` z plikiem, a następnie przy

pomocy metody `Serialize` obiektu klasy `BinaryFormatter` zapiszemy do pliku wcześniej utworzoną tablicę. Zwróćmy uwagę, że serializujemy od razu całą tablicę. W bloku `finally` musimy zamknąć obiekt klasy `FileStream`. Przykładowa implementacja jest zamieszczona poniżej:

```
Osoba[] osoby = {
 new Osoba(1912,"password"){Imie="Alan", Nazwisko="Turing"},
 new Osoba(1903,"password"){Imie="John", Nazwisko="Neumann"},
 new Osoba(1923,"password"){Imie="Edgar", Nazwisko="Codd"}
};

BinaryFormatter formatter = new BinaryFormatter();
FileStream fs = null;
try{
 fs = new FileStream("osoby.dat", FileMode.Create);
 formatter.Serialize(fs, osoby);
}
catch (Exception ex){
 Console.WriteLine("Uwaga wyjątek: {0}!!!", ex.Message);
}
finally{
 if (fs != null)
 fs.Close();
}
```

#### *Test deserializacji binarnej*

Sprawdźmy poprawność zapisu przez odczytanie danych z wcześniej zapisanego pliku. Utwórzmy nową tablicę osób. W bloku `try` utworzymy obiekt klasy `FileStream` powiązany z wcześniej zapisanym plikiem. Przy pomocy metody `Deserialize` obiektu klasy `BinaryFormatter` odczytajmy z pliku tablicę osób. W bloku `finally` zamknijmy strumień plikowy. Na zakończenie programu wypiszmy wartości pól elementów tablicy. Przykładowa implementacja jest zamieszczona poniżej:

```
Osoba[] osoby2 = null;
try{
 fs = new FileStream("osoby.dat", FileMode.Open);
 osoby2 = (Osoba[]) formatter.Deserialize(fs);
}
catch (Exception ex){
 Console.WriteLine("Uwaga wyjątek: {0}!!!", ex.Message);
}
finally{
 if (fs != null)
 fs.Close();
}

foreach (Osoba o in osoby2){
 Console.WriteLine("Imię: {0}, nazwisko: {1}, ile by miał lat: {2}",
 o.Imie,o.Nazwisko,o.Wiek);
 Console.WriteLine("\thasło password: {0}", o.SprawdzHaslo("password"));
 Console.WriteLine("\thasło {1}{2}: {0}",
 o.SprawdzHaslo(o.Imie+o.Nazwisko),o.Imie,o.Nazwisko);
}
```

Zwróćmy uwagę, że pole `hasło` w wyniku serializacji i deserializacji zostało zmienione.

Oba gotowe powyższe projekty znajdują się w katalogu **Demo\Modul14**.

## Serializacja XML

Naszym zadaniem jest utworzenie klasy reprezentującej towar. O towarze potrzebne nam są następujące informacje:

- nazwa
- cena detaliczna i cena hurtowa
- unikalny identyfikator towaru

Klasa musi być serializowalna do postaci XML. Pole reprezentujące unikalny identyfikator towaru powinno być zapisane jako atrybut. Pole reprezentujące cenę hurtową powinno zostać pominięte. Nazwa elementu dla pola reprezentującego nazwę towaru powinna brzmieć `NazwaTowaru`.

### Implementacja serializacji XML w klasie Towar

Załóżmy, że mamy następującą początkową definicję klasy `Towar`:

```
public class Towar{
 public string Nazwa { set; get; }
 public decimal CenaDetaliczna { set; get; }
 public decimal CenaHurtowa { set; get; }
 public int IdTowaru { set; get; }
}
```

Dodajmy do klasy publiczny bezparametrowy konstruktor i nadajmy wszystkim właściwościom domyślne wartości:

```
public Towar(){
 Nazwa = "Brak";
 cenaDetaliczna = 0;
 CenaHurtowa = 0;
 IdTowaru = -1;
}
```

Ponieważ atrybuty związane z serializacją XML zawarte są w przestrzeni nazw `System.Xml.Serialization`, powinniśmy ją zaimportować.

Chcemy, aby właściwość `CenaHurtowa` nie była serializowana, dlatego dodajemy do niej atrybut `XmlIgnoreAttribute`:

```
[XmlIgnore]
public decimal CenaHurtowa { set; get; }
```

Do właściwości `Nazwa` dodajmy atrybut `XmlElementAttribute`, aby zmienić nazwę elementu reprezentującego tę właściwość w dokumencie XML:

```
[XmlElement("NazwaTowaru")]
public string Nazwa { set; get; }
```

Do właściwości `IdTowaru` dodajmy atrybut `XmlAttributeAttribute`, aby właściwość ta była zapisywana jako atrybut elementu, a nie podelement:

```
[XmlAttribute("Id")]
public int IdTowaru { set; get; }
```

### Utworzenie programu testowego

#### Test serializacji XML

W metodzie `Main` utworzymy tablicę towarów i inicjalizujemy te obiekty znanymi sobie wartościami. Utworzymy następnie obiekt klasy `XmlSerializer` oraz zmienną klasy `FileStream`, która będzie reprezentować strumień, do którego prześlemy utworzone towary. W bloku `try` skojarzymy zmienną klasy `FileStream` z plikiem, a następnie zapiszmy do pliku wcześniej utworzoną tablicę przy pomocy metody `Serialize` obiektu klasy `XmlSerializer`. W bloku `finally` zamknijmy



obiekt klasy `FileStream`. Pamiętajmy, że klasa `XmlSerializer` zawarta jest w przestrzeni nazw `System.Xml.Serialization`, w zestawie `System.Xml`. Przykładowa implementacja jest zamieszczona poniżej:

```
Towar[] oferta = {
 new Towar(){Nazwa="Jabłko", CenaDetaliczna=2.5m,
 CenaHurtowa=1.2m, IdTowaru=1},
 new Towar(){Nazwa="Śliwki", CenaDetaliczna=3.2m,
 CenaHurtowa=1.5m, IdTowaru=3},
 new Towar(){Nazwa="Truskawki", CenaDetaliczna=3.8m,
 CenaHurtowa=1.6m, IdTowaru=4}
};

XmlSerializer serializer = new XmlSerializer(typeof(Towar[]));
FileStream fs = null;
try{
 fs = new FileStream("Oferta.xml", FileMode.Create);
 serializer.Serialize(fs, oferta);
}
catch (Exception ex){
 Console.WriteLine("Uwaga wyjątek: {0}!!!", ex.Message);
}
finally{
 if (fs != null)
 fs.Close();
}
```

#### *Test deserializacji XML*

Sprawdźmy poprawność zapisu przez odczytanie danych z wcześniej zapisanego pliku. Utwórzmy nową tablicę towarów. W bloku `try` utworzymy obiekt klasy `FileStream` powiązany z wcześniej zapisanym plikiem. Przy pomocy metody `Deserialize` obiektu klasy `XmlSerializer` odczytajmy z pliku tablicę towarów. W bloku `finally` zamknijmy strumień plikowy. Na zakończenie programu wypiszmy, jakie towary udało się nam odzyskać. Przykładowa implementacja jest zamieszczona poniżej:

```
Towar[] kopiaOferty = null;
try{
 fs = new FileStream("Oferta.xml", FileMode.Open);
 kopiaOferty = (Towar[])serializer.Deserialize(fs);
}
catch (Exception ex){
 Console.WriteLine("Uwaga wyjątek: {0}!!!", ex.Message);
}
finally{
 if (fs != null)
 fs.Close();
}

foreach (Towar t in kopiaOferty){
 Console.WriteLine("Towar {0}, cena detaliczna - {1}, ",
 t.Nazwa, t.CenaDetaliczna);
 Console.WriteLine("cena hurtowa - {0}, id towaru - {1}",
 t.CenaHurtowa, t.IdTowaru);
}
```

Zauważmy że cena hurtowa wszystkich towarów w tablicy `kopiaOferty` ma wartość zero, taką jaką nadaliśmy jej w konstruktorze bezparametrowym.

Oba gotowe powyższe projekty znajdują się w katalogu **Demo\Modul14**.

## Porady praktyczne

- Serializacja XML domyślnie zapisuje wszystkie publiczne pola i publiczne właściwości do odczytu i zapisu. Nie zapisuje niepublicznych właściwości i pól.
- W przypadku serializacji XML nie jest zapisywana informacja o typie, więc w wyniku deserializacji możemy otrzymać obiekt zupełnie innego typu.
- Serializację XML stosuj wtedy, kiedy zamierzasz wymieniać informacje zawarte w obiektach Twojej klasy między programami działającymi na różnych platformach. Serializacja XML zapewnia przenośność.
- Domyślnie klasa `XmlSerializer` nie serializuje pól typu referencyjnego, które zawierają wartość `null`.
- Serializacja XML ignoruje atrybuty związane z serializacją binarną i SOAP.
- Serializację XML możesz również dostosować do własnych potrzeb przez implementację interfejsu `System.Xml.Serialization.IXmlSerializable` w klasie, której obiekty mają być serializowane do dokumentu XML. Interfejs ten zawiera trzy metody:
  - `GetSchema` – metoda obecnie jest zastrzeżona, powinna zwracać wartość `null`
  - `ReadXml` – metoda wywoływana podczas deserializacji
  - `WriteXml` – metoda wywoływana podczas serializacji
- Typy, których obiekty chcesz serializować do dokumentu XML, muszą spełniać następujące warunki:
  - muszą być publiczne
  - muszą posiadać konstruktor bezparametrowy, może być to również konstruktor automatycznie wygenerowany przez kompilator
  - wszystkie ich składowe, które mają być serializowane, muszą być publicznie dostępne
- Typy, których obiekty chcesz serializować binarnie i do protokołu SOAP muszą spełniać następujące warunki:
  - muszą być oznaczone atrybutem `SerializableAttribute`
  - jeśli dziedziczą po innej klasie, musi ona również być serializowalna
  - wszystkie ich pola, które mają być serializowane, muszą być typu serializowalnego
- Serializowane są tylko składowe niestaticzne. Składowe statyczne nie podlegają automatycznemu procesowi serializacji zarówno binarnej, jak i SOAP czy XML.
- Deserializacja binarna i do protokołu SOAP nie wykorzystuje konstruktora bezparametrowego. W przypadku gdy istnieją pola, które nie są serializowane, musisz nadać im wartość ręcznie przez dostosowanie procesu deserializacji do własnych wymagań.
- Następujące zmiany definicji klasy są dopuszczalne, aby można było deserializować binarnie (SOAP) obiekty zserializowane przed modyfikacją klasy:
  - Zmiana kolejności składowych.
  - Zmiana typu pola pod warunkiem, że istnieje konwersja ze starego typu na nowy typ. Proces deserializacji może wykorzystać interfejs `System.IConvertible` do konwersji w przypadku, gdy stary typ implementuje go.
  - Zmiana sygnatury dowolnej metody (również konstruktora).
  - Dodanie lub usunięcie metody.
- Możesz serializować binarnie (przy pomocy obiektu klasy `BinaryFormatter`) kolekcje generyczne pod warunkiem, że typ elementu kolekcji jest typem serializowalnym.
- Dodając nowe pole do typu przez siebie definiowanego opatrz go atrybutem `OptionalFieldAttribute`, aby nie było problemu z jego deserializacją. Podczas deserializacji temu polu zostanie nadana wartość domyślna odpowiednia dla typu danego pola.

## Uwagi dla studenta

Jesteś przygotowany do realizacji laboratorium jeśli:

- Rozumiesz pojęcie serializacji.
- Wiesz do czego służy serializacja XML.
- Potrafisz definiować klasy serializowalne do dokumentu XML.
- Wiesz jak serializować obiekty do pliku XML.
- Umiesz dostosować do własnych wymagań proces serializacji do dokumentu XML za pomocą atrybutów serializacji XML.
- Wiesz do czego służy serializacja binarna.
- Potrafisz definiować klasy serializowane binarnie.
- Wiesz jak serializować obiekty do postaci binarnej.
- Umiesz dostosować do własnych wymagań proces serializacji binarnej.

## Dodatkowe źródła informacji

1. Jesse Liberty, *C#. Programowanie*, Helion, 2005

Książka skierowana do programistów chcących nauczyć się programować w języku C#.

2. Andrew Troelsen, *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition*, Apress, 2007

Książka przeznaczona dla bardziej zaawansowanych programistów. Czwarte wydanie opisuje język C# 3.0 i platformę .NET 3.5.

3. Francesco Balena, Giuseppe Dimauro, *Practical Guidelines and Best Practices for Microsoft Visual Basic .NET and Visual C# Developers*, Microsoft Press, 2005

Książka nie jest podręcznikiem do nauki języka, ale zawiera wiele praktycznych rad jak powinniśmy pisać swoje programy.

4. Codeguru, <http://www.codeguru.pl>

Portal polskiej społeczności programistów .NET. Jeśli nie jesteś tam zarejestrowany, to zarejestruj się koniecznie.

5. C Sharp Tutorial, [http://www.meshplex.org/wiki/C\\_Sharp\\_Tutorial](http://www.meshplex.org/wiki/C_Sharp_Tutorial)

Internetowy kurs języka C#.

6. C# Corner, <http://www.csharpcorner.com>

Portal poświęcony programowaniu w języku C#.

7. Kurs C#, cz. I, [http://www.centrumxp.pl/dotNet/20,1,kategoria,Kurs\\_C\\_cz\\_I.aspx](http://www.centrumxp.pl/dotNet/20,1,kategoria,Kurs_C_cz_I.aspx)

Przystępny kurs języka C# w języku polskim.

## Laboratorium podstawowe


### Problem 1 (czas realizacji 35 min)

Jesteś właścicielem niewielkiej firmy programistycznej wykonującej różnego rodzaju zlecenia. Właśnie dzisiaj przyszedł do Ciebie Twój dobry kolega, właściciel komis samochodowego i poprosił o przygotowanie programu wspierającego jego działalność. Program powinien umożliwić dodawanie kolejnych samochodów i zapisywanie aktualnej listy dostępnych samochodów do pliku. Informacje o samochodzie, które interesują Twojego kolegę, to: marka, producent, rok produkcji, cena zakupu, cena sprzedaży oraz poprzedni właściciel (imię, nazwisko i numer telefonu). Dodatkowo informacja o poprzednim właścicielu ze względu na „ochronę danych osobowych”

powinny być zapisane w oddzielnym pliku w bezpiecznej lokalizacji, a plik zawierający informacje o samochodach powinien zawierać tylko numer ID właściciela.

| Zadanie                                                 | Tok postępowania                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. Utwórz nowy projekt w Visual C# 2008 Express Edition | <ul style="list-style-type: none"> <li>• Otwórz Visual C# 2008 Express Edition.</li> <li>• Z menu wybierz <b>File -&gt; New Project</b>.</li> <li>• Z listy <b>Visual Studio installed templates</b> wybierz <b>Class Library</b>.</li> <li>• W polu <b>Name</b> wpisz <b>KomisBiblioteka.cs</b>.</li> <li>• Kliknij <b>OK</b>.</li> <li>• Z menu wybierz <b>File -&gt; Save KomisBiblioteka</b>.</li> <li>• W polu <b>Location</b> wybierz folder w którym będzie zapisany projekt.</li> <li>• Zaznacz pole wyboru <b>Create directory for solution</b>.</li> <li>• W polu <b>Solution Name</b> wpisz <b>Modul14</b>.</li> <li>• Naciśnij przycisk <b>Save</b>.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                |
| 2. Zaimplementuj klasę Osoba według wymagań             | <ul style="list-style-type: none"> <li>• Zmień nazwę pliku <b>Class.cs</b> na <b>Osoba.cs</b>, klikając w okienku <b>Solution Explorer</b> dany plik i wybierając z menu kontekstowego <b>Rename</b>.</li> <li>• W okienku dialogowym naciśnij przycisk <b>Tak</b>, aby zmienić wszystkie odwołania do elementu <b>Class</b>.</li> <li>• Do klasy <b>Osoba</b> dodaj następujące pola: <b>imie</b>, <b>nazwisko</b> i <b>numerTelefonu</b> typu <b>string</b> oraz <b>idOsoby</b> typu <b>int</b>. Do każdego pola dodaj odpowiadającą mu publiczną właściwość do odczytu i zapisu. Dla pola <b>idOsoby</b> właściwość powinna być tylko do odczytu. Każdej nowo utworzonej osobie przyporządkuj jako <b>idOsoby</b> kolejny numer:</li> </ul> <pre> public class Osoba{     public string Imie { set; get; }     public string Nazwisko { set; get; }     public string NumerTelefonu { set; get; }      private static int nrOsoby = 1;     private int idOsoby = nrOsoby++;     public int IdOsoby { get { return idOsoby; } } } </pre> |
| 3. Uczyń klasę Osoba serializowalną binarnie            | <ul style="list-style-type: none"> <li>• Na górze pliku <b>Osoba.cs</b> zaimportuj przestrzeń nazw <b>System.Runtime.Serialization</b>:<br/> <code>using System.Runtime.Serialization;</code></li> <li>• Do klasy <b>Osoba</b> dodaj atrybut <b>SerializableAttribute</b>:<br/> <code>[Serializable]</code><br/> <code>public class Osoba</code></li> <li>• Do klasy <b>Osoba</b> dodaj metodę, która będzie wywoływana po deserializacji każdego obiektu. W metodzie tej kontroluj wartość statycznego pola <b>nrOsoby</b> oraz sprawdź, czy czasem już nie istnieje osoba o takim numerze ID. Gdy wykryjesz nieodpowiednie ID, rzuć wyjątek:<br/> <code>[OnDeserialized]</code><br/> <code>private void PoDeserializacji(StreamingContext context){</code><br/> <code>    if (idOsoby &lt; nrOsoby)</code><br/> <code>        throw new Exception("Id osoby musi być wartością unikalną!!!");</code><br/> <code>    nrOsoby = idOsoby + 1;</code><br/> <code>}.</code></li> </ul>                                                        |

|                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>4. Dodaj do projektu klasę <b>Samochod</b> tak, aby zawierał wszystkie wymagane pola</p> | <ul style="list-style-type: none"> <li>Wybierz <b>Project</b> -&gt; <b>Add Class</b>.</li> <li>Z listy <b>Visual Studio installed templates</b> wybierz <b>Class</b>.</li> <li>W polu <b>Name</b> wpisz <b>Samochod.cs</b>.</li> <li>Kliknij <b>OK</b>.</li> <li>Uczyń klasę <b>Samochod</b> klasą publiczną. Do klasy <b>Samochod</b> dodaj następujące pola: <b>marka</b>, <b>producent</b> typu <b>string</b>, <b>rokProdukcji</b> typu <b>int</b>, <b>cenaZakupu</b>, <b>cenaSprzedazy</b> typu <b>decimal</b> oraz <b>poprzedniWlasciciel</b> typu <b>Osoba</b>. Do każdego pola dodaj mu odpowiadającą publiczną właściwość do odczytu i zapisu:</li> </ul> <pre>public class Samochod{     public string Marka { set; get; }     public string producent { set; get; }     public int RokProdukcji { set; get; }     public decimal CenaZakupu { set; get; }     public decimal CenaSprzedazy { set; get; }     public Osoba PoprzedniWlasciciel { set; get; } }</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <p>5. Uczyń klasę <b>Samochod</b> serializowalną binarnie</p>                               | <ul style="list-style-type: none"> <li>Na górze pliku <b>Samochod.cs</b> zaimportuj przestrzeń nazw <b>System.Runtime.Serialization</b>:<br/> <pre>using System.Runtime.Serialization;</pre> </li> <li>Do klasy <b>Samochod</b> dodaj atrybut <b>SerializableAttribute</b>:<br/> <pre>[Serializable] public class Samochod</pre> </li> <li>Do klasy <b>Samochod</b> dodaj publiczne statyczne pole <b>PoprzedniWlasciele</b> typu <b>List&lt;Osoba&gt;</b>. Pole to będzie zawierało wszystkich dostępnych właścicieli:<br/> <pre>public static List&lt;Osoba&gt; PoprzedniWlasciele;</pre> </li> <li>W klasie <b>Samochod</b> zaimplementuj interfejs <b>ISerializable</b>. Zachowaj i odczytaj bez żadnych modyfikacji wszystkie pola poza polem <b>PoprzedniWlasciciel</b>. W metodzie <b>GetObjectData</b> zachowaj tylko <b>IdOsoby</b> poprzedniego właściciela. W konstruktorze <b>Samochod(SerializationInfo info, StreamingContext context)</b> sprawdź, czy w dynamicznej tablic <b>PoprzedniWlasciele</b> znajduje się właściciel, którego ID jest równe wczytanemu ID. Jeżeli właściciel istnieje, ustaw go jako poprzedniego właściciela wczytywanego samochodu. Jeżeli nie ma właściciela o żądanym ID, rzuć wyjątek:</li> </ul> <pre>public class Samochod: ISerializable{     ...     public void GetObjectData(SerializationInfo info,         StreamingContext context){         info.AddValue("Marka", Marka);         info.AddValue("Producent", Producent);         info.AddValue("RokProdukcji", RokProdukcji);         info.AddValue("CenaZakupu", CenaZakupu);         info.AddValue("CenaSprzedazy", CenaSprzedazy);         info.AddValue("Id", PoprzedniWlasciciel.IdOsoby);     }      protected Samochod(SerializationInfo info,         StreamingContext context){         Marka = info.GetString("Marka");         Producent = info.GetString("Producent");         RokProdukcji = info.GetInt32("RokProdukcji");         CenaZakupu = info.GetDecimal("CenaZakupu");     } }</pre> |

|                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                      | <pre> CenaSprzedazy = info.GetDecimal("CenaSprzedazy"); int id = info.GetInt32("Id"); foreach (Osoba o in PoprzedniWlasciciele) {     if (o.IdOsoby == id){         PoprzedniWlasciciel = o;         break;     } } if (PoprzedniWlasciciel == null)     throw new Exception("Nie ma właściciela o podanym id"); } } </pre> <ul style="list-style-type: none"> <li>• Dodaj publiczny bezparametrowy konstruktor, aby można było tworzyć obiekty klasy <b>Samochod</b> w łatwy sposób: <pre> public Samochod(){ } </pre> </li> <li>•  Czy konstruktor bezparametrowy jest konieczny w przypadku serializacji binarnej?</li> </ul>                                                                                                                                                                                                                                                                                                                                                        |
| 6. Dadaj nowy projekt do rozwiązania | <ul style="list-style-type: none"> <li>• Kliknij prawym klawiszem myszy w okienku <b>Solution Explorer</b> element reprezentujący rozwiązanie i wybierz z menu kontekstowego <b>Add -&gt; New Project</b>.</li> <li>• Z listy <b>Visual Studio installed templates</b> wybierz <b>Console Application</b>.</li> <li>• W polu <b>Name</b> wpisz <b>Komis</b>.</li> <li>• Kliknij <b>OK</b>.</li> <li>• Kliknij prawym klawiszem myszy w okienku <b>Solution Explorer</b> element reprezentujący nowo utworzony projekt i wybierz z menu kontekstowego <b>Set as StartUp Project</b>.</li> <li>• Kliknij prawym klawiszem myszy w okienku <b>Solution Explorer</b> element reprezentujący nowo utworzony projekt i wybierz z menu kontekstowego <b>Add Reference</b>. W okienku dialogowym <b>Add Reference</b> na zakładce <b>Projects</b> zaznacz <b>KomisBiblioteka</b> i naciśnij klawisz <b>OK</b>.</li> </ul>                                                                                                                                                        |
| 7. Zaimplementuj klasę Program       | <ul style="list-style-type: none"> <li>• Na górze pliku Program.cs zaimportuj bibliotekę <b>KomisBiblioteka</b>: <pre> using KomisBiblioteka; </pre> </li> <li>• Do klasy <b>Program</b> dodaj dwa pola statyczne, jedno reprezentujące samochody w komisie, drugie reprezentujące właścicieli: <pre> static List&lt;Samochod&gt; samochody; static List&lt;Osoba&gt; wlasciciele; </pre> </li> <li>• Do klasy <b>Program</b> dodaj metodę statyczną wyświetlającą dostępne opcje w programie: <pre> static void Menu(){     Console.WriteLine("\n\n\n\n\n");     Console.WriteLine("\t\tA - Dodaj samochod\n");     Console.WriteLine("\t\tB - Wypisz wszystkie samochody\n");     Console.WriteLine("\t\tK - Koniec\n"); } </pre> </li> <li>• Do klasy <b>Program</b> dodaj metodę statyczną dodającą kolejny samochód: <pre> static void DodajSamochod(){     string marka, producent, imie, nazwisko, telefon;     int rok;     decimal cenaZakupu, cenaSprzedazy;      Console.Write("Podaj markę samochodu: ");     marka = Console.ReadLine(); </pre> </li> </ul> |

```

Console.Write("Podaj producenta samochodu: ");
producent = Console.ReadLine();

Console.Write("Podaj cenę zakupu: ");
cenaZakupu = decimal.Parse(Console.ReadLine());

Console.Write("Podaj ceną sprzedaży: ");
cenaSprzedazy = decimal.Parse(Console.ReadLine());

Console.Write("Podaj rok produkcji: ");
rok = int.Parse(Console.ReadLine());

Console.Write("Podaj imię właściciela: ");
imie = Console.ReadLine();

Console.Write("Podaj nazwisko właściciela: ");
nazwisko = Console.ReadLine();

Console.Write("Podaj telefon właściciela: ");
telefon= Console.ReadLine();

Osoba o = new Osoba() { Imie = imie, Nazwisko = nazwisko,
 NumerTelefonu = telefon };
wlascciele.Add(o);

Samochod s = new Samochod(){CenaSprzedazy = cenaSprzedazy,
 CenaZakupu = cenaZakupu, Marka = marka,
 PoprzedniWlasciciel = o,Producent = producent,
 RokProdukcji = rok

};
samochody.Add(s);
}

```

- Do klasy **Program** dodaj metodę statyczną dodającą wyświetlającą samochody dodane do komisju:

```

static void WyswietlSamochody(){
 int i = 1;
 foreach (Samochod s in samochody){
 Console.WriteLine("{0}\t{1}, {2}, {3}, {4:C}, {5:C}",
 i, s.Producent, s.Marka, s.RokProdukcji, s.CenaZakupu,
 s.CenaSprzedazy);
 Console.WriteLine("\t{0} {1}, {2}", s.PoprzedniWlasciciel.Imie,
 s.PoprzedniWlasciciel.Nazwisko,
 s.PoprzedniWlasciciel.NumerTelefonu);
 if (i % 12 == 0)
 Console.ReadKey(true);
 i++;
 }
}

```

- Zaimplementuj metodę **Main**:

```

char c;
do{
 Menu();
 c = Console.ReadKey(true).KeyChar;
 switch (c){
 case 'a':
 case 'A': DodajSamochod();
 break;
 case 'b':
 case 'B': WyszwietlSamochody();
 break;
 }
}

```

|                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                             | <pre> } while (!(c == 'k'    c == 'K')); </pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 8. Dodaj do programu możliwość zapisywania listy samochodów do pliku i wczytywania listy samochodów z pliku | <ul style="list-style-type: none"> <li>Na górze pliku <b>Program.cs</b> zaimportuj konieczne biblioteki: <pre> using System.IO; using System.Runtime.Serialization.Formatters.Binary; </pre> </li> <li>Dodaj metodę wczytującą dane z pliku: <pre> static void Wczytaj(){     if (!File.Exists("Osoby.dat")    !File.Exists("Samochody.dat")){         wlasciciele = new List&lt;Osoba&gt;();         samochody = new List&lt;Samochod&gt;();         Samochod.PoprzedniWlasciciele = wlasciciele;         return;     }      FileStream fs = null;     BinaryFormatter formatter = new BinaryFormatter();     try{         fs = new FileStream("Osoby.dat", FileMode.Open);         wlasciciele = (List&lt;Osoba&gt;)formatter.Deserialize(fs);         Samochod.PoprzedniWlasciciele = wlasciciele;         fs.Close();         fs = new FileStream("Samochody.dat", FileMode.Open);         samochody = (List&lt;Samochod&gt;)formatter.Deserialize(fs);     }     finally{         if (fs != null)             fs.Close();     } } </pre> </li> <li>Wywołaj metodę zdefiniowaną w poprzednim kroku na początku metody <b>Main</b> (przed pętlą <b>do...while</b>): <pre> static void Main(string[] args) {     Wczytaj();     ... } </pre> </li> <li>Dodaj metodę zapisującą dane do pliku: <pre> static void Zapisz(){     FileStream fs = null;     BinaryFormatter formatter = new BinaryFormatter();     try{         fs = new FileStream("Osoby.dat", FileMode.Create);         formatter.Serialize(fs, wlasciciele);         fs.Close();         fs = new FileStream("Samochody.dat", FileMode.Create);         formatter.Serialize(fs, samochody);     }     finally{         if (fs != null)             fs.Close();     } } </pre> </li> <li>W metodzie <b>Menu</b> wyświetl nową pozycję menu, umożliwiającą zapis danych do pliku: <pre> static void Menu(){     ...     Console.WriteLine("\t\tS - Zapisz do pliku\n");     ... } </pre> </li> </ul> |



|                             |                                                                                                                                                                                                                                                                      |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                             | <ul style="list-style-type: none"> <li>W metodzie <b>Main</b> dodaj obsługę dodanej ostatnio pozycji menu: <pre>static void Main(string[] args){     ...     case 's':     case 'S': Zapisz();                break;</pre> </li> </ul>                               |
| 9. Zbuduj i uruchom program | <ul style="list-style-type: none"> <li>Z menu <b>Build</b> wybierz <b>Build Solution</b>. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program.</li> <li>W celu uruchomienia programu, z menu <b>Debug</b> wybierz <b>Start Debugging</b>.</li> </ul> |

## Problem 2(czas realizacji 10 min)

Po pewnym czasie spotkałeś starego znajomego - kolegę od komis samochodowego. Stwierdził on to dobrze by było, aby program zapisywał informację o samochodach w pliku XML. Ułatwiłoby to mu przedstawienie oferty handlowej w Internecie. Dane, które powinny być umieszczone w pliku XML, to: marka, producent, rok produkcji, cena sprzedaży, z tym że rok produkcji musi być umieszczony jako atrybut, a nazwa elementu reprezentującego cen sprzedaży ma być po prostu cena. Postanowiłeś pomóc - czego się nie robi dla przyjaciół.

| Zadanie                                                                                 | Tok postępowania                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. Zmodyfikuj klasę Samochod, aby do dokumentu XML były serializowane tylko żądane pola | <ul style="list-style-type: none"> <li>Przejdź do pliku <b>Samochod.cs</b>.</li> <li>Na górze pliku zaimportuj przestrzeń nazw <b>System.Xml.Serialization</b>. <pre>using System.Xml.Serialization;</pre> </li> <li>Zaznacz, że właściwości nie <b>CenaZakupu</b> i <b>PoprzedniWlasciciel</b> nie będą serializowane do postaci XML. <pre>[XmlIgnore] public decimal CenaZakupu { set; get; }  [XmlIgnore] public Osoba PoprzedniWlasciciel { set; get; }</pre> </li> <li>Zaznacz, że właściwość <b>RokProdukcji</b> będzie serializowana jako atrybut. <pre>[XmlAttribute] public int RokProdukcji { set; get; }</pre> </li> <li>Ustal nazwę elementu XML dla właściwości <b>CenaSprzedazy</b> na <b>Cena</b>. <pre>[XmlElement("Cena")] public decimal CenaSprzedazy { set; get; }</pre> </li> </ul> |
| 2. Zmodyfikuj program główny tak, aby uwzględnił zapis do pliku XML                     | <ul style="list-style-type: none"> <li>Przejdź do pliku <b>Program.cs</b>.</li> <li>Na górze pliku zaimportuj przestrzeń nazw <b>System.Xml.Serialization</b>. <pre>using System.Xml.Serialization;</pre> </li> <li>Do metody <b>Menu</b> dodaj kod wyświetlający nową pozycję menu - zapis do pliku XML. <pre>static void Menu() {     ...     Console.WriteLine("\t\tX - Zapis do pliku XML\n");      Console.WriteLine("\t\tK - Koniec\n"); }</pre> </li> <li>Dodaj metodę, która będzie zapisywać informacje o samochodach do pliku XML.</li> </ul>                                                                                                                                                                                                                                                  |

|                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                | <pre>private static void ZapiszDoXml() {     XmlSerializer serializer =         new XmlSerializer(typeof(List&lt;Samochod&gt;));     using( FileStream fs =         new FileStream("Oferta.xml", FileMode.Create)) {         serializer.Serialize(fs, samochody);     } }</pre> <ul style="list-style-type: none"><li>• Do metody <b>Main</b> dodaj kod uwzględniający nową pozycję menu.</li></ul> <pre>static void Main(string[] args) {     ...     case 'x':         case 'X': ZapiszDoXml();                   break;     ... }</pre> |
| 3. Skompiluj i uruchom program | <ul style="list-style-type: none"><li>• Z menu <b>Build</b> wybierz <b>Build Solution</b>. Jeżeli kompilator wykrył błędy, popraw je i ponownie zbuduj program.</li><li>• W celu uruchomienia programu z menu <b>Debug</b> wybierz <b>Start Debugging</b>.</li></ul>                                                                                                                                                                                                                                                                       |

## Laboratorium rozszerzone

### Zadanie (czas realizacji 90 min)

Twoja firma wygrała przetarg. Przedmiotem przetargu było opracowanie i stworzenie systemu komputerowego wspierającego przeprowadzanie egzaminu. System będzie składał się z następujących programów:

- Programu dla osoby tworzącej egzamin. Będzie tylko jeden program, który będzie działał pod systemem Windows.
- Programu dla osoby egzaminowanej. Osoby zdające egzamin mogą korzystać z różnych programów, działających na różnych platformach. Twoja firma ma opracować program działający pod systemem Windows, korzystając z biblioteki .NET Framework.
- Programu dla koordynatora egzaminu. Będzie tylko jeden program, który będzie działał pod systemem Windows.
- Programu dla osoby sprawdzającej egzamin. Egzaminatorzy mogą korzystać z różnych programów, działających na różnych platformach. Twoja firma ma opracować program działający pod systemem Windows, korzystając z biblioteki .NET Framework.

Osoba tworząca egzamin przygotowuje zestaw pytań wraz z odpowiedziami lub wskazówkami do odpowiedzi (pytania mogą być również otwarte) i wysyła je do koordynatora egzaminu, w naszym przypadku może być to umieszczenie pliku egzaminu w odpowiednim katalogu. Koordynator wczytuje przygotowany egzamin, a następnie rozsyła go (umieszcza plik egzaminu w odpowiednim katalogu) do osób zdających egzamin. Oczywiście treść egzaminu wysłanego do osób egzaminowanych nie zawiera odpowiedzi i wskazówek do pytań. Osoba zdająca egzamin po udzieleniu odpowiedzi na pytania zawarte w egzaminie przesyła rozwiązany egzamin z powrotem do koordynatora (umieszcza plik egzaminu w odpowiednim katalogu). Dodatkowo w pliku egzaminu zawarta jest informacja o osobie zdającej: imię, nazwisko i adres. Następnie koordynator przesyła rozwiązany egzamin do egzaminatora (umieszcza plik egzaminu w odpowiednim katalogu), oczywiście bez informacji na temat osoby egzaminowanej, ale z odpowiedziami i wskazówkami do pytań. Po sprawdzeniu egzaminu egzaminator zwraca sprawdzony egzamin do koordynatora wraz z uwagami i uzasadnieniem oceny poszczególnych pytań. Koordynator rozsyła wyniki egzaminu do osób egzaminowanych.

Twoim zadaniem jest opracowanie koncepcyjnego systemu oraz napisanie odpowiednich programów (tylko tych części, które są wymagane do wczytania/ zapisu odpowiednich dokumentów z/do pliku).