Platforma .NET – Wykład 4
Programowanie w C# – Część 2/3

Osoba prowadząca wykład, laboratorium i projekt:
dr hab. inż. Marek Sawerwain, prof. UZ

Instytut Sterowania i Systemów Informatycznych
Uniwersytet Zielonogórski

e-mail : M.Sawerwain@issi.uz.zgora.pl
tel. (praca) : 68 328 2321,
pok. 328a A-2,
ul. Prof. Z.Szafrana 2,
65-246 Zielona Góra

Ostatnia kompilacja pliku: Monday 5th June, 2023, t: 23:04

Spis treści

1. Wprowadzenie
   - Plan wykładu
2. Wstęp do języka C# – część II
   - Model obiektowy
   - Konstruktor i destruktor
   - Właściwości
   - Dziedziczenie
   - Klasa podstawowa – klasa object
   - Interfejs
   - Rozszerzenia i ułatwienia
3. Typy uogólnione i λ-wyrażenia
   - Typy uogólnione
   - Lambda wyrażenia
4. Już za tydzień na wykładzie

Plan wykładu – spotkania tydzień po tygodniu

(1) Informacje o wykładzie, pojęcie platformy, podstawowe informacje o platformie .NET

(2) Składowe platformy .NET: CLR, CTS, języki programowania, biblioteki klas, pojęcie podzespołu (ang. assembly)

(3) Programowanie w C# – środowisko VS, MonoDevelop, syntaktyka C#, wyjątki, współpraca z DLL

(4) Programowanie w C# – model obiektowy, typy uogólnione, lambda wyrażenia

(5) Programowanie w C# – aplikacje „okienkowe", programowanie wielowątkowe

(6) Programowanie w F# – podstawy, przetwarzanie danych tekstowych,

(*) "Klasówka I", czyli egzamin cześć pierwsza

(7) Dostęp do baz danych

Plan wykładu – tydzień po tygodniu

(8) Język zapytań LINQ, Entity Framework
(9) Obsługa standardu XML
(10) Technologia ASP.NET 1/2
(11) Technologia ASP.NET 2/2
(12) Model widok i kontroler – Model View Controller
(13) Tworzenie usług sieciowych SOAP i WCF (komunikacja sieciowa)
(14) Wykład monograficzny .NET 1
(15) Wykład monograficzny .NET 2
(*) "Klasówka II", czyli egzamin cześć druga

Plan wykładu

1. Wszystko co trzeba o wiedzieć o C#, a boimy się zapytać – część II,
   1. model obiektowy,
   2. obiekty, klasy i struktury,
   3. dziedziczenie, słowo kluczowe sealed
2. typy uogólnione
   1. rola typów uogólnionych,
   2. przykład typów uogólnionych,
3. lambda wyrażenia,
   1. definicja lambda-wyrażeń,
   2. proste przykłady lambda-wyrażeń.

Filary programowania zorientowanego obiektowo (OOP)

Wszystkie zorientowane obiektowo języki programowania (Smalltalk, Objective-C, C#, Java, Visual Basic, C++, etc.), zawsze odnoszą się do następujących trzech pojęć, bardzo często nazywanymi filarami programowania zorientowanego obiektowo:

(I) ukrywanie danych – encapsulation, w jaki sposób język ukrywa detale implementacji oraz dba o spójność danych?,

(II) dziedziczenie – inheritance: w jaki sposób promowane jest ponowne użycie kodu?,

(III) polimorfizm – polymorphism: w jaki sposób język pozwala traktować relacje pomiędzy obiektami.

Model obiektowy widziany przez pryzmat CTS'u

Pojęcie typu jest bardzo często odnoszone do pojęcia reprezentacji danych, szczególnie w przypadku języków zorientowanych na wartość. W przypadku języków obiektowych typ odnosi się raczej do zachowania niż do reprezentacji danych. W przypadku CTS stosowane pojęcie typu, stanowi kompromis pomiędzy podejściem zorientowanym na wartość i zachowanie. Znajduje to odniesienie np.: w przypadku porównywania typów, dwa typy są kompatybilne wtedy i tylko wtedy jeśli mają kompatybilne reprezentacje i zachowanie. Wobec tego, w przypadku CTS, jeśli jeden typ jest wyprowadzany z innego typu, to instancja typy wyprowadzanego może zostać podstawiona do typu bazowego, ponieważ zarówno reprezentacja i zachowanie są kompatybilne.
Dodatkowo, jeśli dwa typy mają zasadniczo inne reprezentacje to są to inne typy. W przypadku niektórych języków programowania obiektowego, zakłada się, że dwa obiekty są tego samego typu, jeśli odpowiadają na pewien zestaw komunikatów w taki sam sposób. To pojęcie, również jest obecne w CTS, bowiem można powiedzieć iż tego typu obiekty implementują ten sam zestaw interfejsów.

Model obiektowy widziany przez pryzmat CTS'u

W przypadku niektórych języków programowania zorientowanych obiektowo takich jak np.: Smalltalk, Objective-C założono iż przekazywanie komunikatów pełni fundamentalną rolę w modelu przetwarzania. W przypadku CTS'u tą rolę pełni pojęcie metody wirtualnej, gdzie sygnatura tej metody pełni rolę komunikatu.
Specyfikacja CTS'u nie oferuje bezpośredniego wsparcia dla programowania bez-typowego – „typeless programming'. Inaczej mówiąc, nie jest możliwe wywołanie nie-statycznej metody bez poznania typu obiektu. Niemniej, technika programowania bez-typowego jest dostępna poprzez mechanizm refleksji, o ile dana implementacja CTSu oferuje tego typu pakiet.

Pojęcie sygnatury

Metody, konstruktor (a dokładniej konstruktorzy instancji), indeksery i operatory są charakteryzowane przez ich sygnaturę:

- sygnatura metody to nazwa, liczba parametrów oraz typ i rodzaj (value, ref, output) jej parametrów, w kolejności od lewego do prawo. Sygnatura metody nie zawiera informacji o typie powrotnym, nie zawiera nazw parametrów lub nazwy typu, nie włączany jest także modyfikator parametrów dla najbardziej zewnętrznego prawego parametru. W przypadku, gdy parametr typu dołącza typ parametru metody, pozycja parametru typu jest stosowana w procesie sprawdzania ekwiwalencji (nazwa typu nie jest brana pod uwagę).
- Sygnatura konstruktora instancji zawiera typ i rodzaj (value, reference, output) każdego parametry formalnego, od lewego do prawego. Sygnatura nie zawiera nazw parametrów i modyfikatorów parametrów, które mogą zostać wyspecyfikowane dla zewnętrznego prawego parametru.
- Sygnatura indeksera zawiera typ i rodzaj (value, reference, output), każdego parametru formalnego, od lewego do prawego. Sygnatura nie zawiera typy elementu lub nazw parametrów i modyfikatorów parametrów które mogą zostać wyspecyfikowane dla zewnętrznego prawego parametru.
- Sygnatura operatora zawiera nazwę operatora i typ każdego parametru formalnego, od lewego do prawego. Sygnatura nie zawiera typu rezultatu i nazw parametrów.

Struktury

Typ strukturalny jest typem wartościowym (a więc wartości są umieszczane na stosie). Ogólnie typ ten jest przeznaczony do opisywania „konkretnych" typów (obiektów) jak np.: pojęcia punktu, prostokąta:

```
public struct Książka
{
    public decimal cena;
    public string tytuł;
    public string autor;
}
```

Punkt widzenia OOP

W metodologii OOP, w kontekście CLR/CTS typ strukturalny to tzw. lekka klasa, bowiem posiada pewne elementy ukrywania danych, jednak nie można tworzyć hierarchii tego typu klas.

Korzystanie z instancji typu

Przykład prostej i zarazem typowej struktury:

```
struct Point {
    public int X;
    public int Y;

    public Point(int XPos, int YPos) {
        X = XPos;
        Y = YPos;
    }

    public void Increment() { X++; Y++; }
    public void Decrement() { X--; Y--; }

    public void Display() {
        Console.WriteLine("X = {0}, Y = {1}", X, Y);
    }
}
```

Przykład użycia:

```
Point myPoint;
myPoint.X = 349; myPoint.Y = 76;
myPoint.Display();
...
Point p2 = new Point(50, 60);
p2.Display();
```

Struktury – uwagi

Struktura (słowo kluczowe struct) posiada taką samą syntaktykę jak klasa, jednak istnieją pewne istotne ograniczenia w stosowaniu, uwagi dot. struktur przedstawiają się następująco:

1. pola struktury nie mogą zostać zainicjalizowane dopóki nie zostaną zadeklarowane jako pola stałe bądź statyczne,
2. struktura nie posiada domyślnego konstruktora (konstruktor bezparametrowy) a także destruktora,
3. wartości typu strukturalnego są powielona w przypadku przypisania, nowo powstałe kopie są niezależne od starszych kopii,
4. struktury są typem wartościowym w odróżnieniu od klasy, która jest typem referencyjnym,
5. w przeciwieństwie do klas, można powoływać nowe instancje bez stosowania słowa kluczowego new,
6. możliwe jest stosowanie konstruktorów z parametrami,
7. struktura nie może dziedziczyć z innej struktury czy klasy, a także nie może stanowić bazy dla innych struktur czy klas, jednak struktura dziedziczy bezpośrednio z **System.ValueType**, która to klasa dziedziczy z **System.Object**.
8. struktury mogą implementować interfejsy,
9. typ strukturalny może przyjmować wartość null.

Składowe klas

Przegląd składników klas:

Typ składowej	Opis
stałe	wartości stałe w klasie
pola	zmienne w klasie
metody	funkcje/akcje/zadanie realizowane przez klasę
własności	akcje/wartości z rozdziałem na odczyt i zapis
indeksy	akcje związane z operatorem dostępu indeksowego
zdarzenia	zdarzenia generowane przez klasę
operatory	konwersje oraz operatory wspierane przez klasę
konstruktor	inicjalizacja składowych obiektu
destruktor	czynności wykonywane przez usunięciem instancji
typy	zagnieżdżone typy zadeklarowane w klasie

Modyfikatory dostępu

Podane modyfikatory dotyczą klas, struktur, interfejsów oraz typu enum:

Typ dostępu	Opis
private	dostęp jest ograniczony do typu
protected	dostęp jest ograniczony do typu oraz typów dziedziczących
public	dostęp nie jest limitowany
internal	dostęp jest ograniczony do podzespołu
internal protected	dostęp jest ograniczony do podzespołu bądź do typów dziedziczących z danego typu

Przestrzenie nazw

Przestrzenie nazw nie posiadają modyfikatorów dostępu.

Modyfikatory dostępu dla typów zagnieżdżonych

Typy deklarowane na najwyższym poziomie, które nie są zagnieżdżone w innych typach mogą posiadać tylko modyfikator internal lub public. Domyślnie dostęp do tego typu typów, to dostęp internal.

Typ	Domyślny dostęp	Inne możliwe rodzaje dostępu
enum	public	brak
class	private	public, protected, internal, private, protected internal
interface	public	brak
struct	private	public, internal, private

Typy zagnieżdżone będące elementami innych typów mogą mieć inny typ dostępu określony przez powyższą tabelę.

Korzystanie z obiektu

Obiekty, czyli instancje klas tworzy się tylko i wyłącznie za pomocą słowa kluczowego **new**:

```
ClassOne varOne;
varOne = new ClassOne();

    albo

ClassOne varOne = new ClassOne();
```

Odwołanie się do pola czy do metody wykorzystuje operator dostępu reprezentowanego przez znak kropki:

```
zmienna1.pole_X = 20;
zmienna1.Metoda1();
```

Konstruktor

Konstruktor, czy to klasy czy struktury zawsze jest wywoływany. Klasa oraz struktura może posiadać wiele konstruktorów różniących się argumentami. Pojęcie konstruktora pozwala na nadanie wartości domyślnych, ograniczenie liczby instancji oraz przyczynia się do tworzenia bardziej elastycznego kodu i łatwiejszego do zrozumienia.
Konstruktor to m.in.:

- metoda wywoływana tuż po utworzeniu obiektu,
- posiada taką samą nazwę jak typ w którym jest zdefiniowany,
- w jego przypadku nie określa się wartości zwracanej,
- wywoływany przez operator **new** (klasy statyczne oraz struktury także mogą posiadać konstruktor),
- wolno go przeciążać.

W przypadku braku konstruktora, język C# zawsze tworzy jeden domyślny konstruktor, którego zadaniem jest nadanie wartości domyślnych wartości wszystkim dostępnym polom w klasie.

Przykłady konstruktorów

Konstruktor domyślny (C# zawsze tworzy konstruktor domyślny, który nadaje polom wartości początkowe) i przeciążony:

```
class ClassName {
    public ClassName() { ... }
    public ClassName(int a1, double a2) { ... }
}
```

Tworzenie obiektów:

```
ClassName x1 = new ClassName( );
ClassName x2 = new ClassName(2, 3.3);
```

Słowo **this** w konstruktorze:

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

Wywołanie konstruktora klasy bazowej:

```
public ClassName(double r) : base(r, 0.0, 1.2) {
...
}
```

Konstruktory prywatne i statyczne

Konstruktor prywatny to specjalna odmiana konstruktora instancji. Jest stosowany w klasach zawierających tylko elementy statyczne. Jeśli klasa zawiera tylko prywatne konstruktory i żadnego konstruktora publicznego, to inne klasy (z wyjątkiem klas zagnieżdżonych) nie mogą tworzyć instancji tej klasy.

```
class NaturalLog {

    private NaturalLog() { }

    public static double e = Math.E;
}
```

Deklaracja pustego konstruktora eliminuje samodzielne tworzenie przez kompilator domyślnego konstruktora.

Konstruktory prywatne i statyczne

Konstruktor statyczny jest stosowany do inicjalizacji dowolnych danych statycznych lub wykonuje akcje szczególne, które powinny być wykonane tylko raz. Konstruktor taki jest wywoływany przed powołaniem pierwszej instancji lub gdy pojawia się referencja do statycznych składowych.

```
class SimpleClass {
    static readonly long baseline;

    static SimpleClass() {
        baseline = DateTime.Now.Ticks;
    }
}
```

Konstruktor statyczny posiada następujące własności:

- modyfikatory dostępu nie są brane pod uwagę i nie posiada parametrów,
- jest wywoływany automatycznie do inicjalizacji klasy przed powołaniem pierwszej instancji lub w przypadku dostępu do statycznych składowych,
- konstruktor statyczny nie może zostać wywołany bezpośrednio,
- użytkownik nie ma kontroli na konstruktorem w momencie jego wywołania przez program,
- przykładem typowego zastosowania konstruktora statycznego jest prosty system logowania, gdzie konstruktor jest użyty do zapisania logu do pliku,
- konstruktor statyczne jest także wykorzystywany do tworzenia klas opakowujących (wrapper class) dla kodu niezarządzanego, w takim przypadku wywołanie konstruktora można wykorzystać do wywołania metody LoadLibrary,
- jeśli konstruktor statyczny zgłasza wyjątek, to wyjątek ten nie będzie zgłaszany ponownie, a tym samym typ pozostaje nie zainicjalizowany w domenie aplikacji w której działa dany program.

Konstruktor kopiujący

W przeciwieństwie do np.: języka C++, język C# nie oferuje i nie tworzy konstruktora kopiującego, rozwiązaniem jest podanie własnej metody tego typu lub utworzenie konstruktora z odpowiednim argumentem:

```
class Osoba {
    private string imię;
    private int wiek;

    public Osoba(Osoba _o) {
        imię = _o.imię;
        wiek = _o.wiek;
    }

    // inna wersja konstruktora, wywołująca
    // tzw. konstruktor instancji
    public Osoba(Osoba _o)
        : this(_o.imię, _o.wiek) {
        ...
    }

    public Osoba(string imię, int wiek) {
        this.imię = imię;
        this.wiek = wiek;
    }
    ...
}
```

Destruktor

Destruktor jest wywoływany w momencie usuwania obiektu przez „garbage collector”.

```
class Pojazd
{
    ~Pojazd() {
        // cleanup statements...
    }
}
```

Uwagi o destruktorze:

- destruktor nie może być stosowany w strukturach,
- klasa posiada tylko jeden destruktor,
- destruktor nie jest dziedziczony ani przeciążany,
- destruktor nie może być wywołany bezpośrednio, jest wywoływany automatycznie przez CLR,
- nie jest możliwe stosowanie modyfikatorów a także destruktor nie posiada argumentów.

Inicjalizatory obiektów

Inicjalizator obiektu umożliwia nadanie wartości publicznym polom i właściwościom bez jawnego wywoływania konstruktura za pomocą notacji stosowanej dla tablicy, dla typu:

```
class Kot {
    public int Wiek;
    public string Imię;
}
```

Prosty przykład:

```
Kot cat = new Kot { Wiek = 10, Imię = "Puszek" };
```

Inicjalizatory obiektów

Inny przykład:

```
class Klasa1 {
    public int a;
    public int b;
}

class Klasa2 {
    public int x; public int y; public Klasa1 K;
    public Klasa2() {
        x = 100; y = 200;
        K = new Klasa1();
        K.a = 10; K.b = 20;
    }
    public Klasa2(int _x, int _y, Klasa1 k)
    {
        x = _x; y = _y; K = k;
    }
}
```

Tworzenie obiektów:

```
Klasa2 x = new Klasa2 { x = 20, K = new Klasa1 { a = 20, b = 30 } };
Klasa1 k1 = new Klasa1() { a = 20, b = 30 };
Klasa2 y = new Klasa2(1, 2, k1) { y = 10, x = 50 };
```

Właściwości

Właściwości/Własności definiują tzw. „wirtualne" atrybuty, przez co możliwa jest publiczna ekspozycja danej wartości, a także ścisłe określenie operacji związanych z atrybutem: nadanie wartości (**set**) i odczytanie (**get**) wartości. Własności pozwalają także na ukrycie implementacji i/lub kodu weryfikującego (własności mogą być wirtualne i naturalnie dziedziczone).

```
[modyfikator dostępu] typ Nazwa {
    get { ... }
    set { ... }
}
```

Kilka uwag o własnościach:

- sekcja **get** jest stosowana do odczytu wartości własności, sekcja **set** określa kod przeznaczony do nadania nowej wartości, możliwe jest także stosowanie różnych modyfikatorów dostępu dla sekcji **set** i **get**,
- słowo kluczowe **value** jest stosowane w sekcji **set** do reprezentacji przypisywanej wartości,
- własności bez sekcji **set** są własnościami tylko do odczytu,
- dla prostych własności, gdzie nie ma potrzebny podawania specjalnego kodu możliwe jest użycie tzw. własności samo-implementujących się (ang. auto-implemented properties).

```
class Czas {
    private int sekundy;

    public int Godziny {
        get { return sekundy / 3600; }
        set { sekundy = value * 3600; }
    }
}
```

Własności asymetrycznym i samo-implementujące się

Przykład o publicznym odczycie ale chronionym zapisie:

```
private string tekst = "Witojcie!!!";
public string Tekst {
    get {
        return tekst;
    }
    protected set {
        tekst = value;
    }
}
```

Samo-implementujące się własności o „trywialnych” sekcjach set i get:

```
class Customer {
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerID { get; set; }
...
}
```

Dostęp indeksowany – indekser

Indeksery pozwalają na dostęp do obiektu klasy lub struktury realizowany przez operator **[ ]** w sposób charakterystyczny dla tablicy. Indeksery to odmiana własności, przy czym sekcje zapisu i odczytu przyjmują dodatkowe parametry. Podstawowe uwagi i własności to min.:

- indeksery pozwalają na wprowadzenie indeksowanie do obiektów w sposób analogiczny do tablic,
- sekcja get odczytuje wartość, natomiast set przypisuje wartość,
- słowo kluczowe this jest stosowne do definicji indekserów,
- słowo kluczowe value jest stosowane do określanie przypisywanej wartości w sekcji set indeksera,
- indeksery nie muszą być indeksowane przez wartości całkowite, co pozwala na tworzenie bardziej wyrafinowanych mechanizmów indeksowania,
- indekser może być przeciążony,
- indekser może posiadać więcej parametrów np.: do realizacji tablic wielowymiarowych.

Przykład indeksera

Przykładowy indekser:

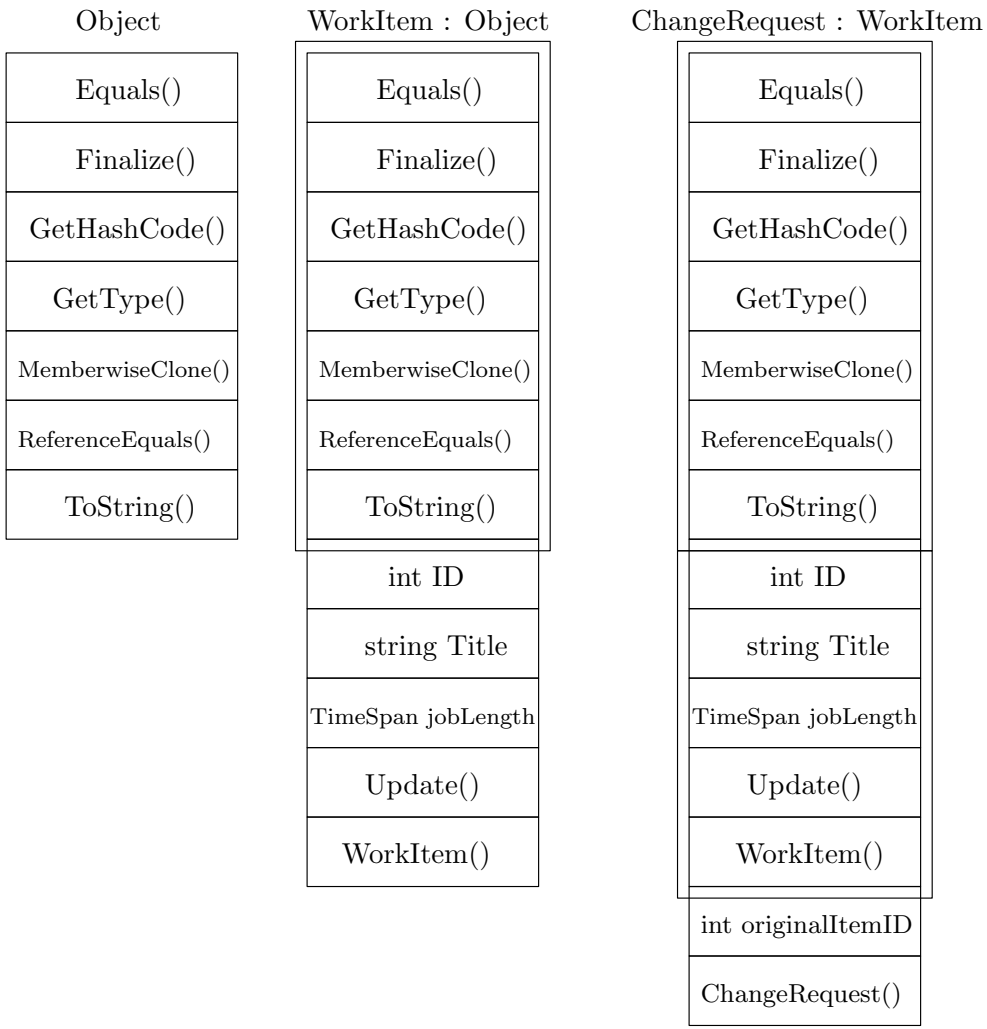
```
public class PowiesciOrazHistorie {
    Historia[] tab;
    public Historia this [int index] {
          get {
                return tab[index];
          }
          set {
                if (value != null) {
                      tab[index] = value;
                }
          }
    }
}
```

Przykład użycia:

```
PowiesciOrazHistorie st = new PowiesciOrazHistorie();
st[155] = new Historia("Historia numer jeden");
st[0] = new Historia("Opowieść historia");
```

Dziedziczenie

Dziedziczenie pozwala na tworzenie klas, które używają, rozszerzają bądź modyfikują zachowanie istniejących klas. Klasa z której dziedziczone są własności, metody, pole to klasa bazowa, a nowo powstała klasa, to klasa pochodna.
Przykład dziedziczenia dla klas object, WorkItem, ChangeRequest:



Dziedziczenie

W przeciwieństwie do C++, język C# wspiera tylko dziedziczenie jednokrotne (bez dodatkowych typów dziedziczenia jak protected i private). Możliwe jest jednak dziedziczenie (implementowanie) z wielu interfejsów. Choć brakuje dodatkowych modyfikatorów dostępu, to jednak możliwe jest ukrywanie/maskowanie poszczególnych pól.

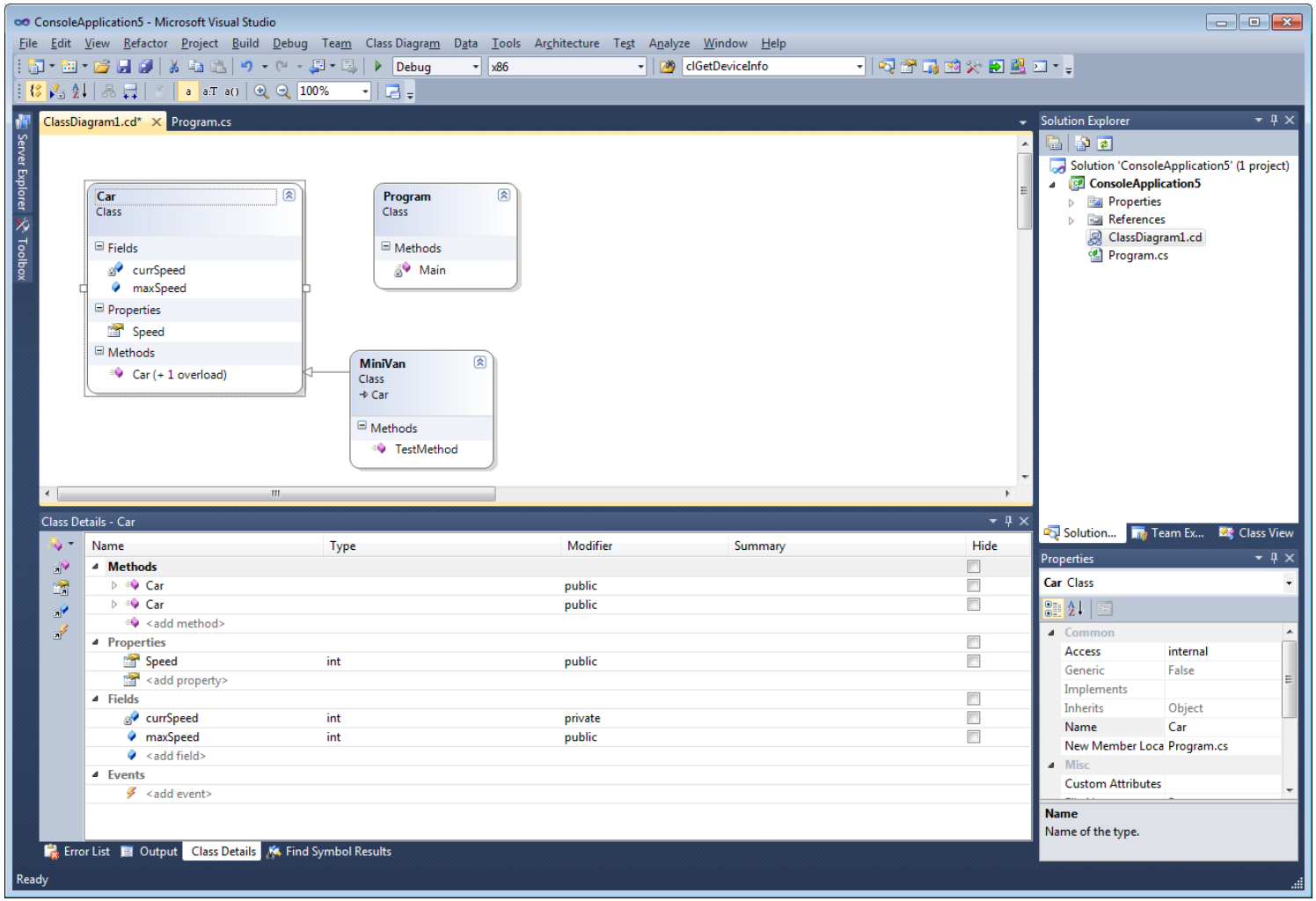
```
class SomeClass {
    public string Field1 = "SomeClass Field1";
    public void Method1(string value) {
        Console.WriteLine("SomeClass.Method1: {0}", value);
    }
}

class OtherClass : SomeClass
{
    new public string Field1 = "OtherClass Field1";
    new public void Method1(string value) {
        Console.WriteLine("OtherClass.Method1: {0}", value);
    }
}
```

Sygnatura

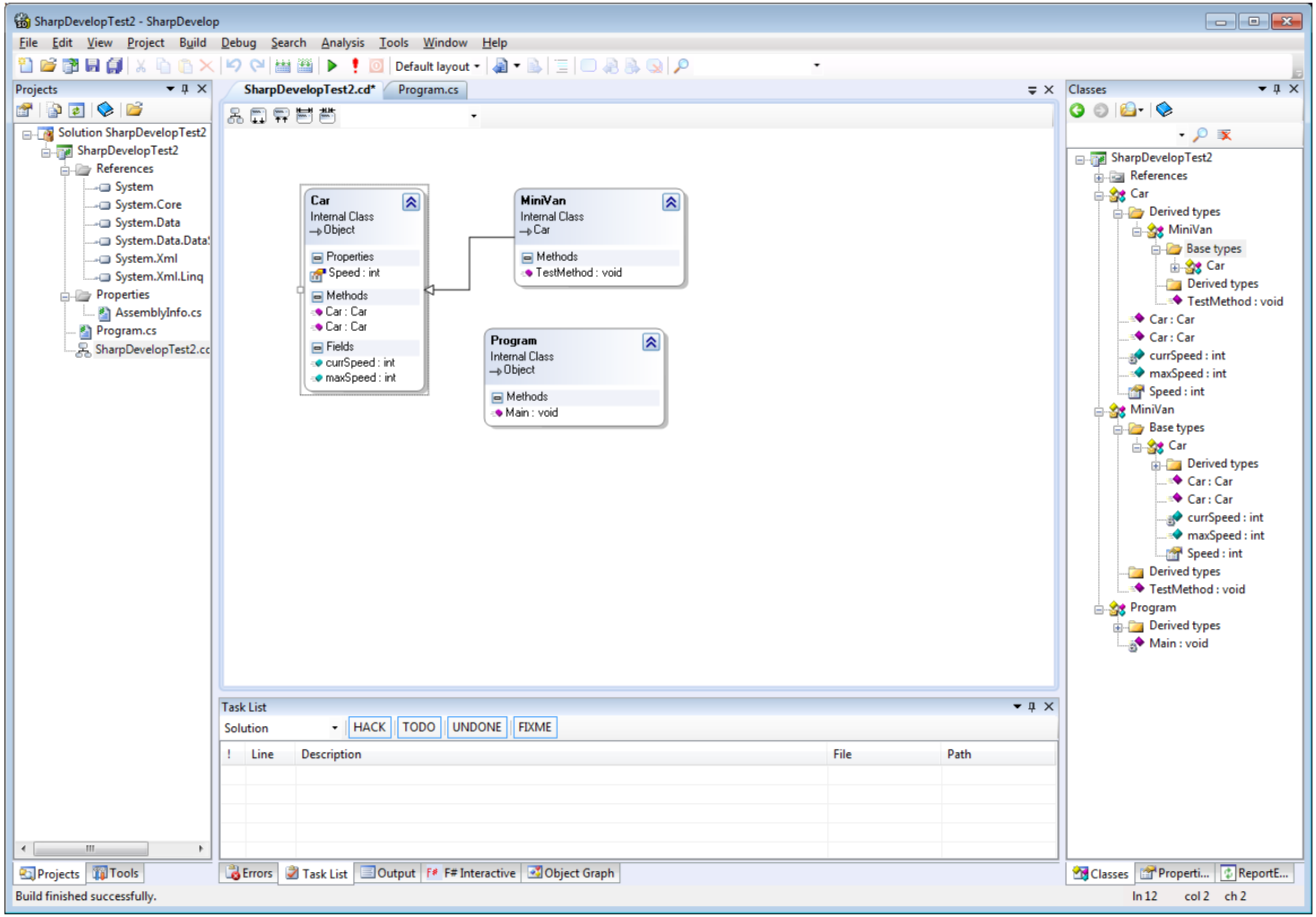
Sygnatura składa się z nazwy, parametrów ale nie należy do niej typ zwracanej wartości.

Dziedziczenie



Visual Studio i narzędzie do modelowania w standardzie UML.

Dziedziczenie



Sharp-Develop i narzędzie do modelowania w standardzie UML.

Słowo kluczowe virtual

Metody wirtualne pozwalają w klasach pochodnych modyfikować zachowanie się składowych w klasach bazowych. Oznaczenie, że składowa ma być wirutalna wymaga zastosowania słowa kluczowego virtual. Nadpisanie składowej wirtualnej wymaga zastosowania słowa kluczowego override.

```
class KlasaBazowa {
    public void Print() {
        Console.WriteLine("print z klasy bazowej");
    }
}

class KlasaPochodna : KlasaBazowa {
    new public void Print() {
        Console.WriteLine("print z klasy pochodnej");
    }
}

class Program {
    static void Main() {
        KlasaPochodna p = new KlasaPochodna();
        KlasaBazowa b = (KlasaBazowa)p;

        p.Print();
        b.Print();
    }
}
```

Metody wirtualne

Następująca modyfikacja dodatnie słów virtual oraz override:

```
class KlasaBazowa {
       virtual public void Print() {
           Console.WriteLine("print z klasy bazowej");
       }
   }
   class KlasaPochodna : KlasaBazowa {
       override public void Print() {
           Console.WriteLine("print z klasy pochodnej");
       }
   }
```

Metoda Print z klasy bazowej zostanie nadpisana metodą Print z klasy pochodnej. Inne uwagi są następujące:

- metody nadpisująca i nadpisana muszą posiadać ten sam modyfikator dostępu,
- nie można nadpisać metody statycznej oraz nie-wirtualnej,
- metody, własności, indeksery oraz zdarzenia mogą być deklarowane jako wirtualne i nadpisywane.

Nowa klasa

Wprowadzenie kolejnej klasy, naturalnie pozwala tylko na zmianę metody podstawowej:

```
class DrugaKlasaPochodna : KlasaPochodna
{
    override public void Print() {
        Console.WriteLine("print z drugiej klasy pochodnej");
    }
}
```

Funkcja main ma postać:

```
DrugaKlasaPochodna p = new DrugaKlasaPochodna();
KlasaBazowa b = (KlasaBazowa)p;

p.Print();
b.Print()
```

Nadpisanie i nowa składowa

Jednak deklaracja tej metody jako nowej:

```
class DrugaKlasaPochodna : KlasaPochodna
{
    new public void Print() {
        Console.WriteLine("print z drugiej klasy pochodnej");
    }
}
```

Spowoduje, że będzie obowiązywać nadpisanie z klasy pochodnej.

Wirtualne własności

Przykład wirtualnej własności:

```
class BaseClass {
    private int _myInt = 5;
    virtual public int Property {
        get { return _myInt; }
    }
}
class DerivedClass : BaseClass {
    private int _myInt = 10;
    override public int Property {
        get { return _myInt; }
    }
}

static void Main() {
    DerivedClass d = new DerivedClass();
    BaseClass b = (BaseClass)d;

    Console.WriteLine(d.Property);
    Console.WriteLine(b.Property);

    Console.ReadLine();
}
```

Słowo kluczowe sealed

Słowo kluczowe sealed użyte dla składowych (metody i własności) oznacza iż nie będzie możliwe ich nadpisanie (zawsze muszą być stosowane ze słowem override):

```
class KlasaBazowa {
    public virtual void f() {
        ...
    };
}
class KlasaPochodna {
    public sealed override void f() {
        ...
    };
}
```

W przypadku klasy oznacza, że nie będzie możliwe dziedziczenie z tej klasy:

```
sealed class Klasa {
    ...
};
```

Oznacza to także, iż wszystkie metody, własności, pola otrzymują status **sealed**.

Klasa abstrakcyjna

Słowo kluczowe **abstract** oznacza iż dana metoda, własności bądź zdarzenie nie będzie zaimplementowana w danym typie. Oznacza to iż dana klasa nie może powołać swojej instancji ale może stanowić bazę dla innej klasy.

```
abstract class ShapesClass {
    abstract public int Area();
}
class Square : ShapesClass {
    int side = 0;

    public Square(int n) {
        side = n;
    }

    public override int Area() {
        return side * side;
    }

    static void Main()  {
        Square sq = new Square(12);
        Console.WriteLine("Area of the square = {0}", sq.Area());
    }
}
```

Klasa **object**

Typ object (alias do typu System.Object) to podstawowa klasa wszystkich innych typów w platformie .NET oraz języku C#. Wszystkie inne typy domyślnie dziedziczą z typu **object**. Cztery najważniejsze publiczne metody to:

1. Equals – porównywanie dwóch obiektów,
2. Finalize – operacje zwalniania zasobów przed ostatecznych usunięciem obiektu,
3. GetHashCode – warto funkcji generującej wartość skrótu, co pozwala na stosowanie obiektu w tablicach otwartych,
4. ToString – reprezentacja obiektu w postaci ciągu znaków.

Również ważne są także następujące chronione metody:

1. GetType() – zwraca obiekt Type, który w pełni opisuje dany typ, jest to związane z technologią RTTI – Runtime Type Identification,
2. MemberwiseClone() – wykonuje kopię obiektu poprzez powielanie wartości poszczególnych składowych.

Przykładowe implementacja metod podstawowych

```
class Point {
    public int x, y;

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

    public override bool Equals(object obj)
    {
        if (obj.GetType() != this.GetType()) return false;

        Point other = (Point) obj;

        return (this.x == other.x) && (this.y == other.y);
    }
    ....
}
```

Porównanie dwóch obiektów typy Point.

Przykładowe implementacja metod podstawowych

```
....
public override int GetHashCode()
{
    return x ^ y;
}

public override String ToString()
{
    return String.Format("({0}, {1})", x, y);
}

public Point Copy()
{
    return (Point) this.MemberwiseClone();
}
```

Implementcja GetHashCode i ToString a także implementacja metody Copy wykorzystująca MemberwiseClone.

Pojęcie interfejsu

Interfejs jest typem referencyjnym, którego zadaniem jest specyfikacja zbioru składowych, jednakże bez ich implementacji, bowiem to zadanie jest przeznaczone dla klasy bądź struktury:

- deklaracja interfejsu nie może zawierać danych,
- deklaracja obiektu może zawierać tylko deklaracje następujących składowych: metody, własności, zdarzenia, indeksery,
- deklaracja składowych nie możne zawierać części implementacyjnych,
- nazwa interfejsu zazwyczaj zaczyna się od dużej litery „I" np.: IComparable, IEnumerable),
- możliwe jest także stosowanie słowa kluczowego partial, do tworzenia interfejsów częściowych.

```
interface IMyInterface {
    int DoStuff ( int nVar1, long lVar2 );
    double DoOtherStuff( string s, long x );
}
```

Interfejs w praktyce

Sortowanie za pomocą metody Sort wymaga zgodności z interfejsem IComparable:

```
class DataClass : IComparable
{
    public int _Value;
    public int CompareTo(object obj) {
        DataClass o = (DataClass)obj;
        if (this._Value < o._Value) return -1;
        if (this._Value > o._Value) return 1;
        return 0;
    }
}
```

Sortowanie elementów typu DataClass:

```
var IntTab = new [] { 22, 3, 1, 3, 7 };
DataClass[] dArr = new DataClass[5];
for (int i = 0; i < 5; i++) {
        dArr[i] = new DataClass();
        dArr[i]._Value = IntTab[i];
    }
...
Array.Sort(dArr);
```

Jawna i niejawna implementacja interfejsu

Technika stosowana gdy w kilku interfejsach znajduje się taka sama nazwa np.: metody:

```
interface IInterOne {
    void f();
}
interface IInterTwo {
    void f();
}

class Klasa : IInterOne, IInterTwo {
    public void f() {
        // Implementacja w sposób niejawny
    }
    void IInterTwo.f() {
        // Implementacja w sposób jawny
    }
}
```

Klasy wieloczęściowe

Możliwe jest rozdzielenie definicji klasy, struktury, interfejsu na dwa lub więcej plików. Każdy plik zawiera tylko fragment definicji type. Wszystkie części definicji typu są łączone dopiero w momencie kompilacji:

1. technika ta jest przydatna w przypadku dużych projektów, bowiem klasa może zostać rozdzielona na kilku programistów którzy mogą pracować nad jedną klasą w tym samym czasie,
2. technika ta jest również wykorzystywana dla automatycznie generowanego kodu źródłowego, dodatkowy kod może być związany z daną klasę bez konieczności tworzenia całego pliku (środowisku VS stosuje tą technikę podczas tworzenia aplikacji Windows Forms oraz w przypadku usług sieciowych),

Do wprowadzenia klasy częściowej stosuje się słowo kluczowe **partial**:

```
public partial class Student
{
    public void ChodźNaWykłady()
    {
    }
}
public partial class Student
{
    public void IdźNaObiad()
    {
    }
}
```

Metody rozszerzające

Metody rozszerzające pozwalają dodawać metody do istniejących typów bez tworzenia nowych typów pochodnych, technika ta także nie wymaga rekompilacji rozszerzanych klas, czy jakiejkolwiek jawnej modyfikacji oryginalnego typu.

```
namespace StringExtensionMethods {
    public static class NewExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                  StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

using StringExtensionMethods;
...
string s = "Hello Extension Methods";
int i = s.WordCount();
```

Typy uogólnione, typy uniwersalne (generics)

Typy uniwersalne mogą być stosowane do implementacji klas, struktur, interfejsów. Głównym parametrem takiego typu jest umieszczony w nawiasach ostrych < ... > inny zdefiniowany typ.

```
class MyStack <T>
{
    int StackPointer = 0;
    T [] StackArray;

    public void Push(T x ) {...}

    public T Pop() {...}
    ...
}
```

Prosta kolekcja

```
class BasicCollection<T> {
    private T[] arr = new T[100];

    public T this[int i] {
        get {
            return arr[i];
        }
        set {
            arr[i] = value;
        }
    }
}

class Program {
    static void Main(string[] args) {
        BasicCollection<string> stringColl = new BasicCollection<string>();

        stringColl[0] = "Hello, World";
        System.Console.WriteLine(stringColl[0]);
    }
}
```

Lambda wyrażenia

Tego typu wyrażenia są szeroko stosowane w LINQ, a są to anonimowe funkcje. Wyrażenie albo anonimowa funkcja/metoda jest określona w następujący sposób:

```
(input parameters) => expression   albo   (input parameters) => { statement; }
```

Przykład operacji binarnej:

```
double op1 = 13.3;
double op2 = 33.4;
double factor = 15.02;
BinaryOperation operation = (x, y) = > {
    Console.WriteLine(f);
    return f * ((x + y) / y);
};
double result = operation(op1, op2);
```

Lambda wyrażenia – przykłady

Liczby, których reszta z dzielenia jest równa jedności:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1)
```

Pierwsza liczba mniejsza niż siedem:

```
var n = numbers.TakeWhile(n => n < 7);
```

Argument równy wartość pięć:

```
Func<int, bool> FiveTest = x => x == 5;
bool result = FiveTest(4);
```

W następnym tygodniu między innymi

1. przeciążanie operatorów i konwersje typów,
2. operatory „is”, „as”, podstawy mechanizmu refleksji,
3. kolekcje,
4. hierarchia wyjątków,
5. zdarzenia oraz delegacje
6. główne klasy i obiekty aplikacji okienkowych,
7. wątki – klasa Thread.

Proponowane tematy prac pisemnych:

1. omówić zagadnienia dziedziczenia prywatnego oraz chronionego, które nie jest wprost oferowane przez język C#,
2. porównanie klasy object i jej roli w językach obiektowych jak np.: Java, Objective-C, Smalltalk,
3. źródła idei lambda-wyrażeń i ich realizacja w C# i innych językach programowania podobnych do C#.

Dziękuje za uwagę!!!