Modelowanie i Programowanie Obiektowe

POLITECHNIKA
POZNAŃSKA

Wykład IV: Platforma .Net

28 październik 2013

Platforma .NET

- Platforma .NET według Microsoft to integralny komponent systemu Windows umożliwiający tworzenie i uruchamianie nowoczesnych aplikacji i usług sieciowych. Cele realizowane w ramach platformy:
  - Dostarcza kompletne zorientowane obiektowo środowisko niezależnie czy obiekt: przechowywany i uruchamiany lokalnie, wykonywany lokalnie i rozproszony w Internecie czy wykonywany zdalnie
  - Środowisko wykonawcze minimalizujące konflikty podczas wytwarzania i wersjonowania oprogramowania
  - Środowisko wykonawcze wspierające bezpieczne i wydajne wykonanie kodu
  - Zapewnia spójność pomiędzy aplikacjami przeznaczonymi na platformę Windows czy opartymi na technologiach Web'owych

Platforma .NET - komponenty

- Wspólne środowisko uruchomieniowe (*ang. Common Language runtime*)
  - Agent zarządzający kodem podczas wykonania
  - Zarządzanie pamięcią
  - Zarządzanie wątkami
  - Zdalność (*ang. remoting*)
  - Bezpieczeństwo (typowanie – CTS – sprawdzanie typów, dokładność, zarządzanie zasobami, rejestrem itd.)
  - Łączenie wielu języków w jednym projekcie (język → narzędzie)
  - Wydajność – kod nie jest interpretowany, a kompilowany podczas wykonania (JIT – just-in-

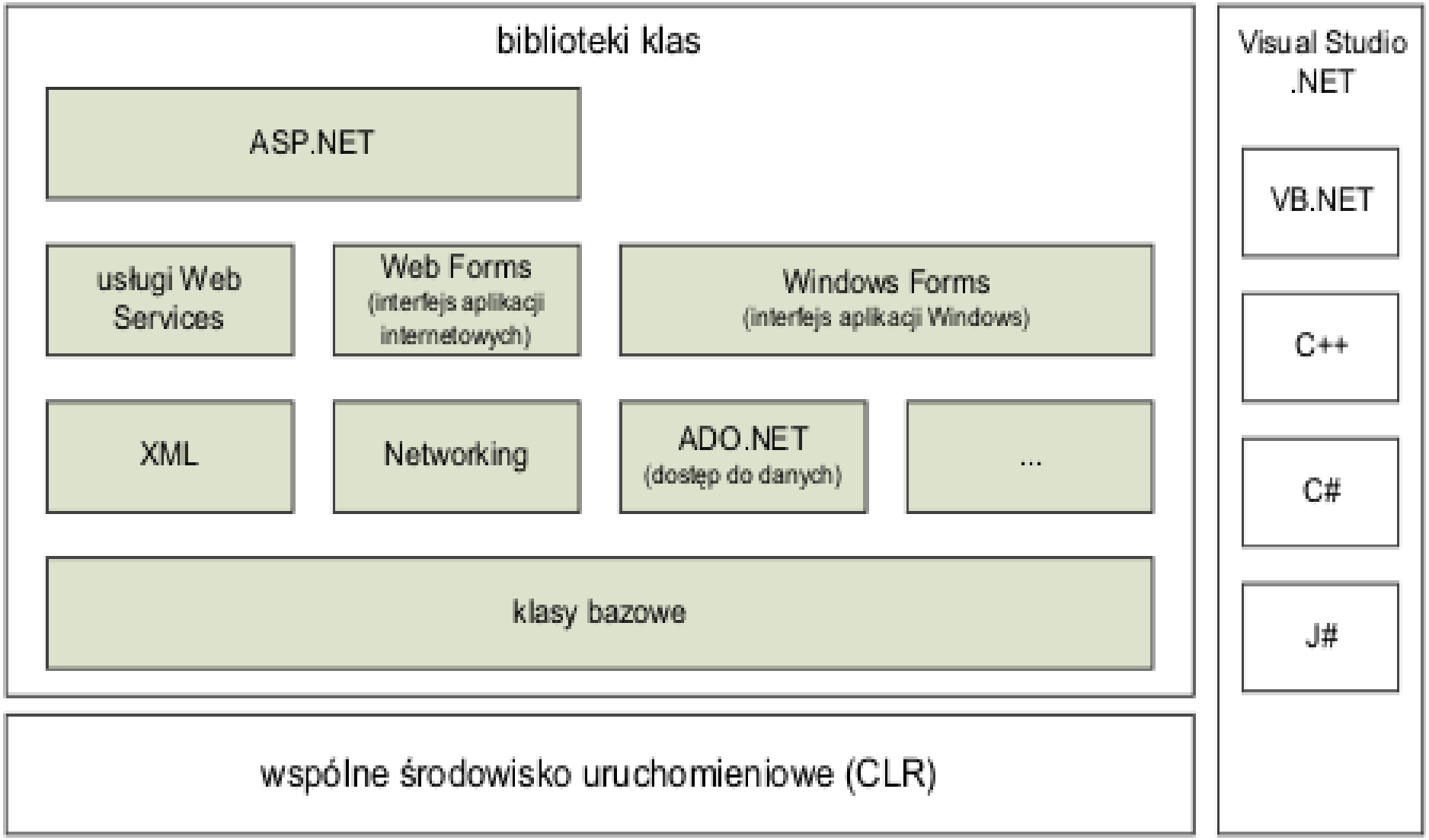
Platforma .NET - komponenty

- .NET Framework class library
  - Kompleksowa, zorientowana obiektowo, re-używalna kolekcja typów zintegrowana z CLR
  - pozwalająca na łatwe pisanie aplikacji ogólnego przeznaczenia:
    - aplikacje konsolowe
    - okienkowe (WinForms, WPF)
    - Web'owe (ASP.NET)
    - Aplikacje sieciowe (WCF)
    - usługi sieciowe (XML Web Services)
    - Procesy biznesowe (WWF)

Kod zarządzany i niezarządzany

- Kod niezarządzany (**ang. unmanaged code**)
  Kod, który wykonywany jest bezpośrednio przez system operacyjny, poza wspólnym środowiskiem uruchomieniowym (CLR) platformy Microsoft .NET Framework. Kod niezarządzany musi dostarczać własne wsparcie dla zabezpieczeń, zarządzania pamięcią oraz sprawdzania typów danych - w przeciwieństwie do kodu zarządzanego, który pobiera te informacje ze wspólnego środowiska uruchomieniowego. Kod niezarządzany musi być wykonany poza platformą .NET Framework.
- Kod zarządzany (**ang. managed code**)
  Kod kompilowany i wykonywany przez platformę Microsoft® .NET Framework, a dokładniej – przez wspólne środowisko uruchomieniowe (CLR). Kod zarządzany musi przekazywać do CLR wszystkie niezbędne informacje, aby możliwe było korzystanie z takich usług, jak zarządzanie pamięcią, integracja między językami, bezpieczeństwo oparte na uprawnieniach kodu (CAS - Code Access Security) oraz automatyczne zarządzanie cyklem życia obiektów. Każdy kod aplikacji oparty na standardowym języku pośrednim Microsoft (MSIL - Microsoft Intermediate Language) wykonywany jest jako kod zarządzany.

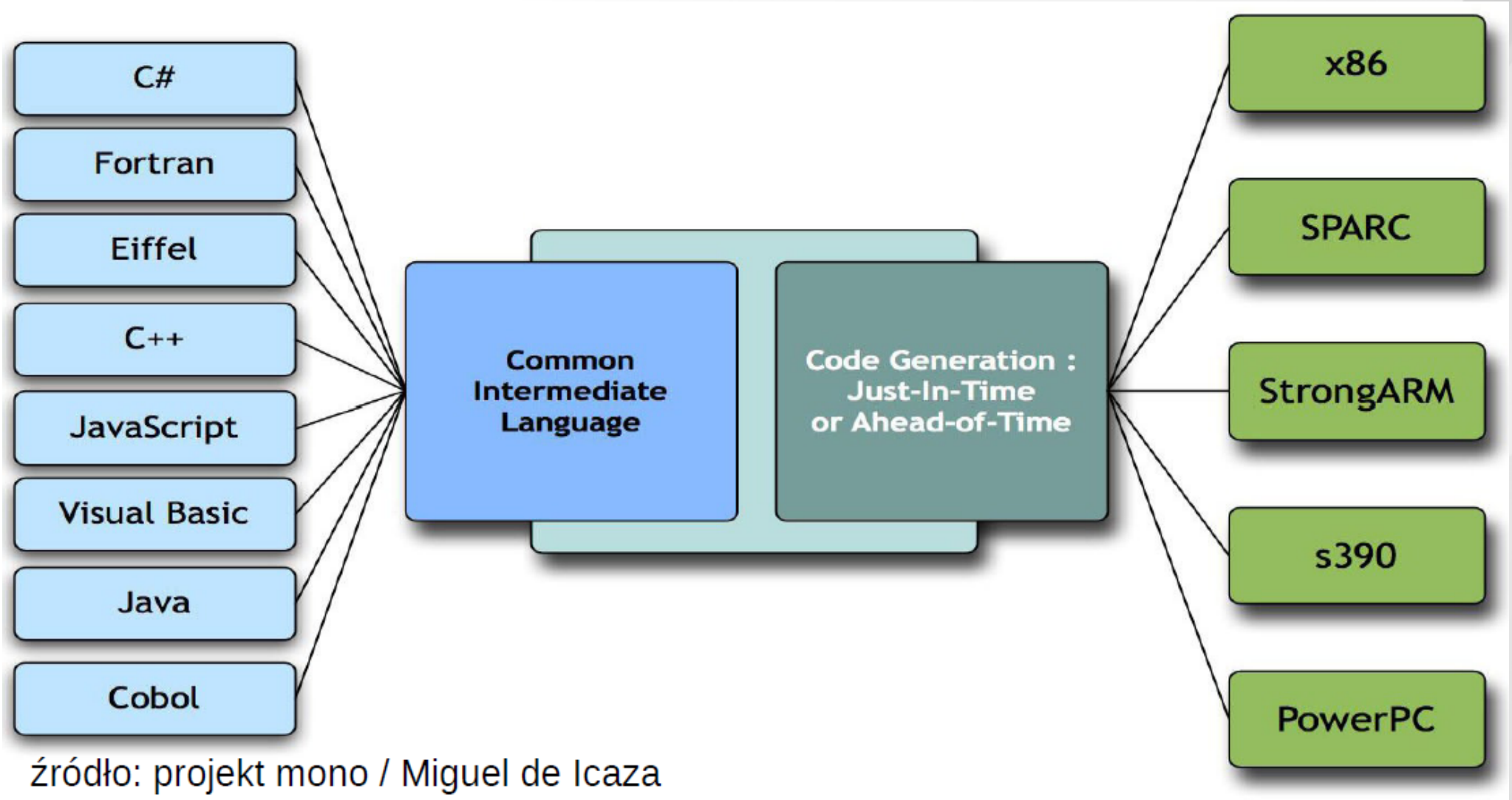
Platforma .NET - komponenty



Platforma .NET – na jednym slajdzie

- Rozwiązanie kompleksowe
  - języki programowania – C#, VB.NET, J#, MC++, ...
  - środowisko uruchomieniowe
  - zbiór bibliotek
  - narzędzia programistyczne
- wieloprocesorowe (i wielosystemowe)
- objęte standardami ECMA* i ISO

Platforma .NET – rys. poglądowy



źródło: projekt mono / Miguel de Icaza

Akronimy

- CLI – Common Language Infrastructure
  - CLR – Common Language Runtime (implementacja CLI przez Microsoft)
  - CTS – Common Type System
  - CIL – Common Intermediate Language
  - CLS – Common Language Specification
  - VES – Virtual Execution System
- JiT, AoT – metody kompilacji

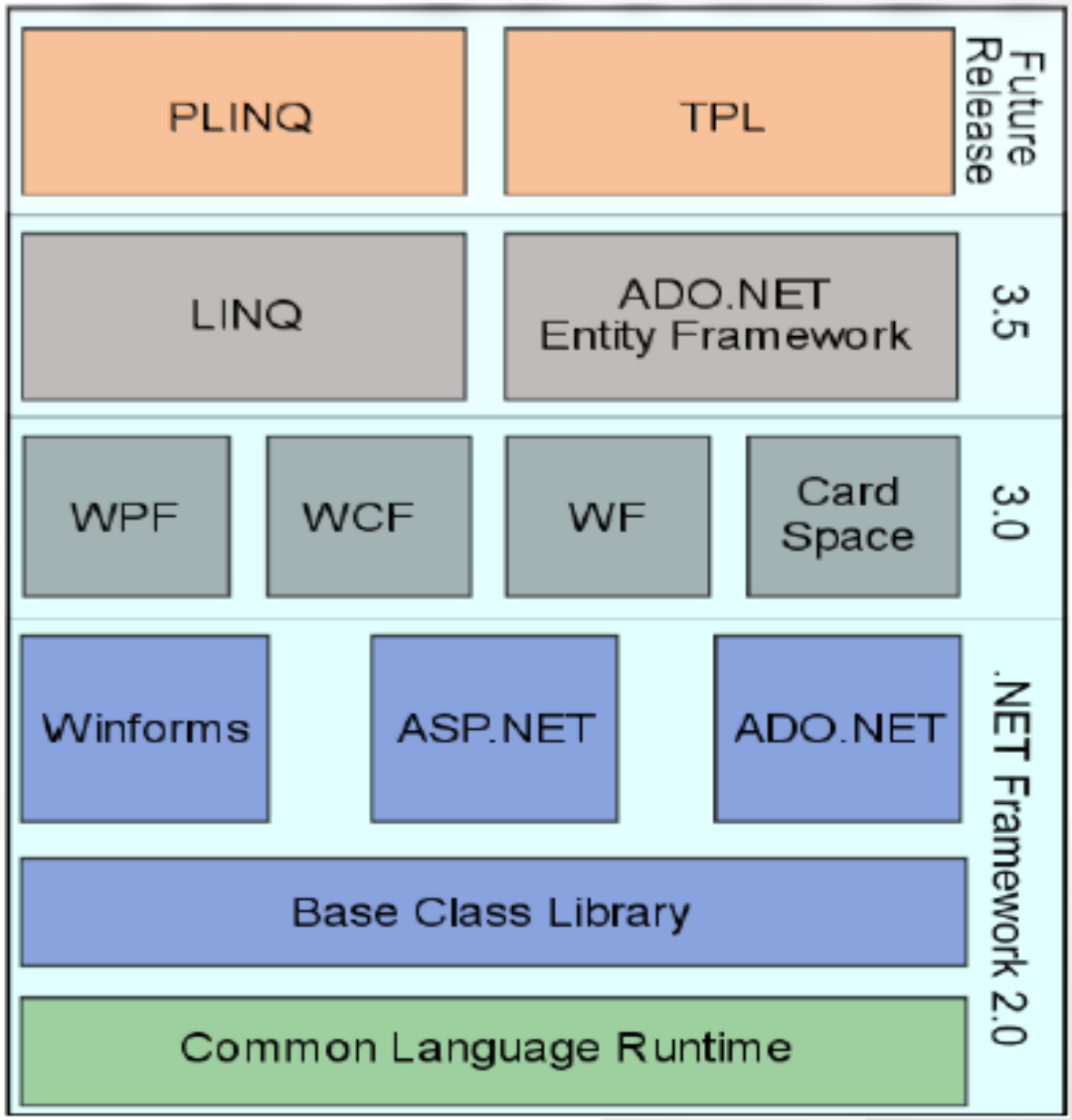
Historia

- dlaczego .NET?
- lipiec 2000 – opracowanie standardów CLI i języka C# przez Microsoft, Hewlett-Packard i Intel
- grudzień 2001 – standaryzacja przez ECMA jako ECMA-334 (C#) oraz ECMA-335 (CLI)
- kwiecień 2003 – standaryzacja przez ISO jako ISO/IEC 23270 (C#) i ISO/IEC 23271 (CLI)

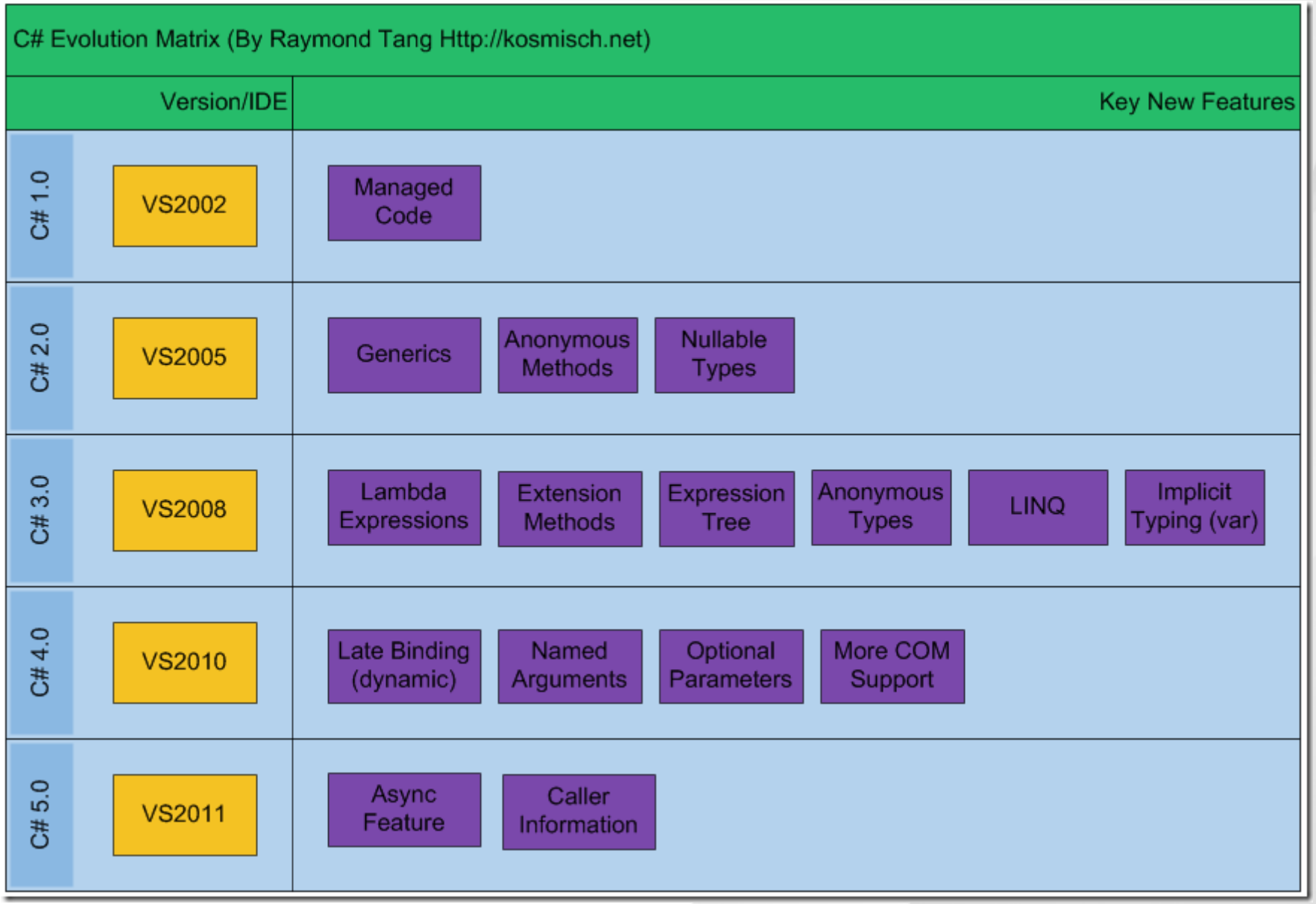
Historia – kolejne wersje platformy

- MS .NET Framework 1.0 – 2002,
- MS .NET Framework 1.1 i Compact Framework 1.0 – 2003
- MS .NET Framework 2.0 i Micro Framework –2005,
- MS .NET Framework 3.0 – 2006
  - Windows Presentation Foundation (WPF, Avalone),
  - Windows Communication Foundation (WCF, Indigo),
  - Windows Workflow Foundation (WF),
  - Windows CardSpace (WCS, InfoCard)
- MS .NET Framework 3.5 – 2007
- MS .NET Framework 3.5 SP1 – 2009
- MS .NET Framework 4.0 – 2010
- MS .NET Framework 4.5 – 2011/2012

Architektura .NET 3.5



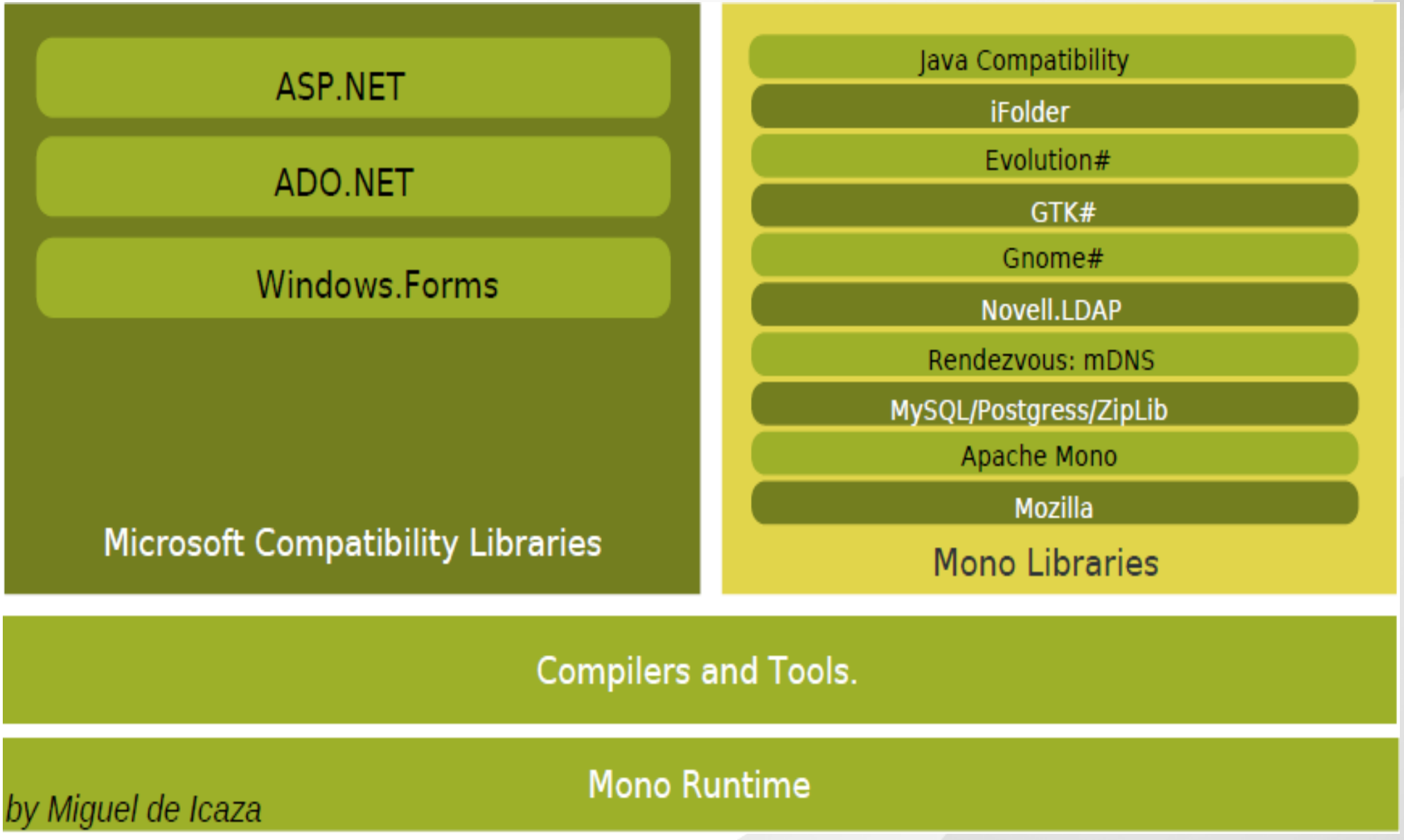
kolejne wersje języka C#



Historia – inne platformy

- Mono (http://www.mono-project.com/)
  - 2001 – ogłoszenie projektu przez Miguela de Icaza
  - 2004 – mono 1.0 (MS .NET Framework 1.1)
  - 2006 – mono 1.2 (MS .NET 1.1 z MWF, .NET 2.0)
  - 2008 – mono 2.0 (C# 3.0, MWF 2.0)
  - 2009 – mono 2.6
  - duża liczba bibliotek OpenSource, w tym poszerzających framework
  - możliwość „pogrzebania w środku”
- DotGNU (http://www.gnu.org/software/dotgnu)

Architektura mono



Specyfikacja platformy .NET

- Kod zarządzany
  - Zarządzanie pamięcią – *ang. Garbage Collector*
  - Zarządzanie aplikacją
  - Zarządzanie bezpieczeństwem
- **asemblacja (***ang. assembly***) –** skompilowany kod do uruchomienia lub wykorzystania:
  - Wykonywane (*ang. process assemblies*) → *.exe
  - Biblioteczne (*ang. library assemblies*) → *.dll
  - Posiadają one kod w CIL
  - CIL kompilowany jest do kodu maszynowego w trakcie kompilacji JIT CLR
- Reflection API

Języki

- w MS .NET dostępne: C#, J#, VB.NET, C++/CLI, F#, JScript
- w innych projektach: Java, Nemerle, Boo, ...
- DLR (Dynamic Language Runtime) – Python, JavaScript (EcmaScript 3.0), Visual Basic i Ruby
- zalecany C#
  - stworzony pod platformę
  - naturalna składnia
  - bardzo podobny do języka Java
  - aktualnie wersja 5.0 języka (z Framework 4.5)

Programowanie w .NET

- Podstawowa biblioteka klas – przestrzenie:
  - *System* – podstawa innych przestrzeni i typy podstawowe,
  - *System.Collections* – typy kontenerowe,
  - *System.Data* – dostęp do baz danych,
  - *System.Data.SQLClient* – provider dla MS SQL,
  - *System.Drawing* – funkcje graficzne biblioteki GDI+,
  - *System.IO* – operacje I/O na systemie plików,
  - *System.Math* – funkcje matematyczne,
  - *System.Reflection* – mechanizm refleksji,

Programowanie w .NET cd.

- Podstawowa biblioteka klas – przestrzenie:
  - *System*.Security – CAS, kryptografia,
  - *System.Threading* – obsługa wątków,
  - *System.Windows.Forms* – GUI,
  - *System.Net.Sockets* – gniazdka sieciowe,
  - *System.XML* – narzędzia do XML,
  - System.Text – łańcuch znaków, kodowanie,
  - dołączanie przestrzeni: dyrektywa `using`

Programowanie w .NET cd.

- Typy danych
  - proste i referencyjne
  - struktury i klasy
- Definicja klasy

```
public class Demo1
 {
   public Demo1(){ }
   public ~Demo1(){ }
   public void Add1(int x,int y){ y += x;}
   public void Add2(int x,ref int y){ y += x;}
   public void Add3(int x,out int y){ y = x;}
 }
```

Programowanie w .NET cd.

- Parametry
  - domyślnie – przez (kopiowaną) referencję,
  - `out` – przekazanie wyniku na zewnątrz,
  - `ref` – wymuszenie przekazania przez referencję,
- Poziomy dostępu
  - `public` - składowe zadeklarowane przy użyciu tego modyfikatora posiadają dostęp publiczny. Dane składowe można zmienić, a metody wywołać z dowolnego miejsca w kodzie programu. Jeśli pisane oprogramowanie jest biblioteką, to metody można także wywoływać z całego kodu wykorzystującego tę bibliotekę;
  - `Protected` - składowe zadeklarowane z użyciem tego modyfikatora mają dostęp chroniony. Dostęp możliwy jest wyłącznie z klasy, w której zostały one zadeklarowane lub z

Programowanie w .NET cd.

- Poziomy dostępu cd.
  - `private` - składowe zadeklarowane przy użyciu słowa kluczowego private mają dostęp prywatny. Są one dostępne wyłącznie z klasy, w której znajduje się ich deklaracja;
  - `internal` (C#) lub `friend`(VB) - Ten typ dostępu nazywany jest dostępem wewnętrznym lub zaprzyjaźnionym. Składowe opatrzone takimi modyfikatorami dostępne są wyłącznie z podzespołu, w którym zadeklarowano klasę zawierającą te składowe.
  - `protected internal` (C#) lub `protected friend`(VB) - Ten typ dostępu jest połączeniem dostępu chronionego z dostępem wewnętrznym lub zaprzyjaźnionym — składowe dostępne są z podzespołu oraz z klas dziedziczących po klasie, w której zostały zadeklarowane.

Programowanie w .NET cd.

- Składowe klasy
- pola, metody, konstruktory/destruktory
- Właściwości

```
public int Member {
    get { returm mem; }
    set { mem = value; }
}
o.Member = 987;
```

- delegacje
- zdarzenia
- atrybuty

Programowanie w .NET cd.

- Składowe klasy
  - statyczne

```
public class X { public static int m_x =
1;}
```

  - Stałe

```
public class Y { public const int PI= 3.1;}
```

- klasy `sealed`

```
public sealed class Y { int PI= 3.1;}
```

  - klasa – nie można z niej dziedziczyć,
  - metoda – nie może być przeciążona,
  - każda struktura jest `sealed`

Programowanie w .NET cd.

- Wyliczenia

```
public enum ProjLevel : byte { Lite = 0x01, Normal = 0x02 }
```

- Rzutowanie
  - `as` – rzutuj i zwróć referencję nowego typu

```
Cat c = animal as Cat;    //can be null
```

  - `is` – sprawdź, czy rzutowanie możliwe *(bool)*

```
if (animal is Cat) {…
```

- Typy proste: `byte`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float`, `double`, `decimal`, `bool`, `char`,
- Typy strukturalne: `Int32`, `Int64`, `Double`

Programowanie w .NET cd.

- `System.String` – referencyjny, ale nietykalny,
- Metody obsługi typów danych
  - `Equals, GetHashCode, GetType, ToString`
  - `boxing, Parse`
- Tablice

```
int[] t= new int[10];
int[,] t= { { 1,2,3 },{ 3,4,5 } };
```

Programowanie w .NET cd.

- Delegacje
  - odpowiedniki wskaźników na metodę

```
public delegate string FancyStringDel (int x);
public string ConvOne (int val) {...}
public void Method () {
    FancyStringDel del = new FancyStringDel
    (ConvOne);
    del (2);
}
```

Programowanie w .NET cd.

- Zdarzenia (*ang. events*)
  - komunikacja wiele-wiele między klasami
  - Zajście jakiejś szczególnej sytuacji (powiadomienie)

```
public delegate void CalcDel(int x);

public event CalcDel CalculationDone;

    ...

CalculationDone += new CalcDel(ShowRes);

CalculationDone -= new CalcDel(ShowRes);

    ...

    CalculationDone (3);
```

Programowanie w .NET cd.

- Kolekcje
  - List<typ> – lista (tablica) o dynamicznym rozmiarze

- Dziedziczenie – tylko publiczne

```
class Truck :Vehicle
    {
       void GoTo(string s){...}
       void GoTo(Point p){...}
    }
```

Programowanie w .NET cd.

- Pojedyncze dziedziczenie
- Przeciążanie metod i operatorów

```
public static Point operator+ (Point p1, Point p2)
    {...}

    ...

    Point x = ...;

    Point y = ...;

    Point p = x + y;
```

Programowanie w .NET cd.

- Przesłanianie i ukrywanie metod

```
public class Car{
    public virtual double GetMaxSpeed (...);
    public bool Stopped (...);
     }
public class SportCar :Car{
    public override double GetMaxSpeed (...);
    public new string Stopped (...);
     }
```

Programowanie w .NET cd.

- Klasa bazowa – składowa `base`
- Klasa abstrakcyjna – modyfikator `abstract`
- Wyjątki:

  `System.ApplicationException`

  `try / catch / finally / throw`

Programowanie w .NET cd.

- Atrybuty
  - udostępniają mechanizm pozwalający na dodawanie metadanych, czyli dodatkowych danych opisujących określone elementy kodu takie jak metody, klasy w postaci np. instrukcji dla kompilatora czy środowiska testowego.
  - po skompilowaniu aplikacji, poza dostępnymi standardowo metadanymi opisującymi każdy pakiet w środowisku .NET widoczne są również te dodatkowe informacje,które zawarte są w atrybutach
  - są sposobem na rozszerzenie zbioru standardowych metadanych z definicji opisujących dany pakiet środowiska .NET

Programowanie w .NET cd.

- Atrybuty
  - Dostęp i odczytanie metadanych jest możliwe dzięki mechanizmowi refleksji, którego działanie w największym skrócie polega na wykorzystaniu dostępu do wspomnianych metadanych
  - tzw. ”deklaratywne” informacje mogą być wykorzystane zarówno podczas prac projektowych nad pakietem, jak i już w trakcie działania danego rozwiązania

Programowanie w .NET cd.

- Atrybuty

```
public class Podatki
{
    [Obsolete("Zamiast PodatekZa2004(), wykorzystaj PodatekZa2005()", true)]
    static double PodatekZa2004(double Dochod)
    {
        //starsza wycofywana metoda
    }

    static double PodatekZa2005(double Dochod)
    {
        //nowa wersja metody liczącej podatki;
    }
}
//..
```

Programowanie w .NET cd.

- Atrybuty

```
Podatki.PodatekZa2004();//wywołanie starszej metody
```

```
'Podatki.PodatekZa2004()' is obsolete: 'Zamiast PodatekZa2004(),
 wykorzystaj PodatekZa2005()'.
```

programista może zasygnalizować, użytkownikowi komponentu (programiście), że już nie powinien posługiwać się metodą znaną z starszych wersji komponentu. Powyższy przykład prezentuje też, w jak łatwy sposób można udzielić innemu programiście wskazówki dotyczącej zalecanego wykorzystania komponentu.

Programowanie w .NET cd.

- Atrybuty wbudowane
  - `Obsolete` wykorzystywany podczas kompilacji kodu, został już przedstawiony wcześniej. Warto jednak dodatkowo wspomnieć, iż atrybut ten niekoniecznie musi generować błąd kompilacji. Poniższy przykład spowoduje pojawienie się ostrzeżenia, zamiast błędu kompilacji.

```
[Obsolete("Przestarzały kod", false)] //ustawienie parametru na false powoduje,
                      //że kompilator zasygnalizuje ostrzeżenie a nie błąd...
void StaryZlyKod() {//...}
```

  - `Conditional` definiujący wykonanie kodu tylko w trybie debug lub release, znalazł swoje zastosowanie w kompilatorze .NET i jest realizowany przez atrybut

```
[System.Diagnostics.Conditional("DEBUG")]
void KodUruchamianyTylkoWDebugu(){ //... }
```

Programowanie w .NET cd.

- Atrybuty definiowane przez użytkownika
  - określenie zakresu wykorzystania atrybutu (`AttributeUsage`)
  - stworzenie klasy zawierającą implementację zachowania dla naszego atrybutu → klasa taka dziedziczy po `System.Attribute`

Programowanie w .NET cd.

- Atrybuty definiowane przez użytkownika

```
//atrybut [AttributeUsage] wywolany z parametrem AttributeTargets.All powoduje,
//że zakresem będą wszystkie elementy kodu,na który nałożymy nasz atrybut
[AttributeUsage(AttributeTargets.All)]
public class ElementInfo : System.Attribute
{
    public ElementInfo(string Name)
    {
        name = Name;
    }
    public string Name
    {
        get { return name; }
    }
    private string name;

}

//Wykorzystanie utworzonej klasy ElementInfo jako atrybutu klasy Examle4Class
[ElementInfo("Klasa Example4")]
public class Examle4Class
{
//Wykorzystanie atrybutu ElementInfo jako atrybutu zmiennej str
    [ElementInfo("String w Example4 - str")]
    private string str;
}
```

Programowanie w .NET cd.

- Atrybuty definiowane przez użytkownika

```
[AttributeUsage(AttributeTargets.Method|AttributeTargets.Struct,
AllowMultiple=false,Inherited=false]
```

  - `AttributeTargets.*` - lista wskazująca na dozwolone wykorzystanie atrybutu
  - `AllowMultiple` – określa czy dany atrybut może być wykorzystany wielokrotnie
  - `Inherited` - określa czy jeśli element klasy opatrzonej atrybutem znalazł się w klasie dziedziczącej po klasie zawierającej atrybut, także będzie powiązany z tym atrybutem

Podstawowe klasy

- System.IO
  - StreamReader/Writer – tekst
  - BinaryReader/Writer – binaria
  - File, Directory, Path – w tym Path.DirectorySeparatorChar
- System.Enviroment(.NewLine)
- System.Console – I/O w trybie tekstowym
- System.Text.RegularExpressions.Regex

Podstawowe interfejsy

- IComparable – sortowanie
- IDisposable – konwencja „sprzątania”, wsparcie ze strony języka przez using

```
using (SqlConnection cn = new SqlConnection ())
{}
```

tłumaczone na:

```
try { cn = new
SqlConnection(connectionString); ...}

finally { if (null != cn) cn.Dispose();}
```

- IEnumerable, IEnumerator – foreach

Programowanie w .NET – C# 2.0

- Metody anonimowe

```
button1.Click += delegate(object s, EventArgs e) {
    MessageBox.Show("You clicked the button”);
};
```

- Iteratory – implementacja *IEnumerable* w 1.1

```
static void Main(string[] args)
{
    NameAndPlaces t = new NameAndPlaces("Jon",
        new string[] { "London","Hereford","Cambridge","Reading" });
    foreach (string x in t)
      {
        Console.WriteLine(x);
        Console.WriteLine("===");
    }
    Console.ReadLine();
}
```

Programowanie w .NET – C# 2.0

- Iteratory – implementacja *IEnumerable* w 1.1

```
public class NameAndPlaces : IEnumerable
{
    string name;
    string[] places;

    public NameAndPlaces(string name, string[] places)
    {
        this.name = name;
        this.places = places;
    }

    public IEnumerator GetEnumerator()
    {
        yield return "My name is " + name;
        yield return "I have lived in:";
        foreach (string place in places)
        {
            yield return place;
        }
    }
}
```

Programowanie w .NET – C# 2.0

- klasy częściowe

```
public partial class PartialClass {...}
```

- operator ?? - definiuje wartość domyślną dla zmiennej mogącej przyjmować wartość `null`

```
// y = x, unless x is null, in which case y = -1.
int y = x ?? -1;
```

- Generics
  - tworzenie klas, metod, struktur z nieznanym, ale ustalanym i sprawdzanym w czasie kompilacji typem
  - idea bazująca na templates z C++

- Nullable types: `int? x = null`

Programowanie w .NET – C# 2.0

- Generics
  - generyczne kolekcje – System.Collections.Generic
    - List <int>,
    - Dictionary <int, KeyValuePair <string, Osoba>>
- najczęściej używane interfejsy są teraz generyczne
- rozszerzenie bibliotek klas (ADO, ASP, System.Net, ...)

```
public class Dictionary<K,V> where K:IComparable
{
   public void Add(K key,V value)
   {
      ...
      if(((IComparable)key).CompareTo(null)< 0){ ...}
   }
}
```

Zmiany w .NET 3.0

- brak zmian w języku i podstawowych bibliotekach
- dodanie zestawy bibliotek *WinFX*
  - *Windows Communication Foundation*
  - *Windows Presentation Foundation*
  - *Windows Workflow Foundation*
  - *Windows CardSpace*

C# 3.0 (.NET Framework 3.5)

- zmienne typowane implicite

```
var orders = new Dictionary<int, Order>();
```

- metody rozszerzające *(extension methods)*
  - Nowe metody dla istniejących typów (skompilowanych)
  - Są to specjalne metody statyczne wykorzystywane dla instancji obiektów
  - W składni nie widać różnicy pomiędzy standardowymi metodami a rozszerzonymi
  - Pierwszy parametr metody poprzedzony słowem kluczowym `this` informuje jakiego typu dotyczy metoda

C# 3.0 (.NET Framework 3.5)

- metody rozszerzające *(extension methods)*

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
…
string s = "Hello Extension Methods";

int i = s.WordCount();
```

C# 3.0 (.NET Framework 3.5)

- Wyrażenia lambda
  - delegat bez nazwy (delegat anonimowy)
  - operacja, bądź ciąg operacji bez nazwy
  - lewa strona wyrażenia lambda to parametry (może ich w ogóle nie być)
  - prawa strona definiuje
  - obie strony są rozdzielone "=>" operacje

C# 3.0 (.NET Framework 3.5)

- Wyrażenia lambda

```
class Program
{
    delegate double PodnoszenieDoKwadratu(double x);
    delegate double Mnozenie(double x, double y);
    delegate string ZwracanieStringa();
    static void Main(string[] args)
    {
        PodnoszenieDoKwadratu p = x => x * x;
        double Liczba = p(1.2d);
        Console.WriteLine(Liczba); // 1.44
        // Musi być nawias jak jest więcej niż 1 argument
        Mnozenie m = (x, y) => x * y;
        Liczba = m(2d, 3d);
        Console.WriteLine(Liczba); // 6
        // Wyrażenie Lambda bez parametrów.
        // Wyrażenie to jest ciągiem poleceń;
        ZwracanieStringa mc = () => { string s = "cześć"; return s; }; // to
samo co:                                                                ()
=> { "cześć" };
        Console.WriteLine(Slowo); // cześć
    }
}
```

C# 3.0 (.NET Framework 3.5)

- Inicjatory obiektów

```
public class Rectangle
{
    Point p1 = new Point();
    Point p2 = new Point();
    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
var r = new Rectangle
{
    P1 = { X = 0, Y = 1 },
    P2 = { X = 2, Y = 3 }
};
```

- Typy anonimowe

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

C# 3.0 (.NET Framework 3.5)

- LINQ: .NET Language-Integrated Query – integracja z SQL i XML

```
from c in customers
where c.City == "London"
select c
// tłumaczone na:
customers.Where(c => c.City == "London")

from c in customers
where c.City == "London"
from o in c.Orders
where o.OrderDate.Year == 2005
select new { c.Name, o.OrderID, o.Total }
//tłumaczone na:
customers.Where(c => c.City == "London").
SelectMany(
    c =>c.Orders.
    Where(
        o => o.OrderDate.Year == 2005).
        Select(o => new { c.Name, o.OrderID, o.Total })
)
```

Środowiska IDE

- MS Visual Studio .NET 2003/2005/2008/2010
- SharpDevelop
- MonoDevelop
- Inne – x-develop, vim ...

Narzędzia

- Microsoft SDK
  - *ilasm/ildasm* – kompilator/disasembler kodu pośredniego,
  - *ngen* – generacja kodu natywnego,
  - *gacutil* – obsługa GAC (Global Assembly Cache),
  - *sn* – generowanie nazw silnych,
- inne
  - *nant* – narzędzie do budowania projektów,
  - *nunit* – testy jednostkowe,
  - *reflector* – disasembler, przeglądarka *assembly*