

Jak działa Microsoft .NET

Łukasz Rajchel

KEN Solutions

29 maja 2009



Abstrakt

- Historia .NET i C#
- Zasada działania .NET
- Assembly
- CIL
- CTS
- CLS
- CLR
- Podstawy programowania w C#

Programowanie przed .NET

- C/Win32
- C++/MFC
- Visual Basic
- Java/J2EE
- COM
- Windows DNA

Programowanie w C/Win32

- skomplikowane i czasochłonne
- problemy z kompatybilnością aplikacji z różnymi wersjami systemu operacyjnego
- ograniczenia języka C (ręczne zarządzanie pamięcią, wskaźniki, brak obiektowości, funkcje globalne...)
- piekło DLL

Programowanie w C++/MFC

- wygodniejsze i szybsze niż C/Win32
- możliwość mieszania kodu obiektowego i proceduralnego
- wielokrotne dziedziczenie
- ograniczenia języka C++ (ręczne zarządzanie pamięcią, wskaźniki, funkcje globalne...)
- piekło DLL

Programowanie w Visual Basicu

- proste tworzenie skomplikowanych UI, serwerów COM...
- brak pełnej obiektowości (klasycznego dziedziczenia, sparametryzowanych konstruktorów...)
- brak wielowątkowości
- brak możliwości tworzenia usług oraz aplikacji konsolowych
- brak wsparcia unicode'u

Programowanie w Javie/J2EE

- pełna obiektowość
- garbage collection
- problematyczna integracja z innymi językami (z założenia aplikacje powinny być tworzone całkowicie w Javie)

Programowanie COM

- aplikacje mogą współpracować z różnymi językami programowania
- problemy z dziedziczeniem
- skomplikowane programowanie i wdrażanie
- piekło DLL

Programowanie Windows DNA

- Windows Distributed interNet Application Architecture
- wymaga używania wielu różnych technologii (ASP, HTML, XML, JScript, VBScript, ActiveX, COM(+), ADO...)
- skomplikowane programowanie i wdrażenie

Założenia .NET

- łatwe programowanie aplikacji Windows
- współpraca z istniejącym kodem (COM, biblioteki C...)
- pełna integracja języków (dziedziczenie, obsługa wyjątków, debugowanie...)
- wspólne środowisko uruchomieniowe dzielone przez wszystkie języki
- bogata biblioteka klas bazowych
- uproszczona praca z COM
- prosty model wdrożeniowy (brak rejestrowania plików binarnych w rejestrze, możliwość używania wielu wersji tych samych *.dll na tej samej maszynie...)

Założenia projektowe C#

- prosty, nowoczesny, obiektowy język ogólnego przeznaczenia
- silnie typowany
- garbage collection
- obsługa internacjonalizacji
- odpowiedni do tworzenia zarówno aplikacji hostowanych jak i systemów wbudowanych
- syntaktycznie czysty jak Java, prosty jak Visual Basic, o możliwościach C++

Najważniejsze cechy C#

- brak globalnych zmiennych oraz funkcji
- zmienne nie mogą wychodzić poza blok, w którym zostały zdefiniowane
- wskaźniki mogą być używane wyłącznie w kodzie oznaczonym jako niebezpieczny (`unsafe`)
- brak ręcznego zarządzania pamięcią
- wielokrotne dziedziczenie nie jest dozwolone, klasy mogą implementować wiele interfejsów
- automatyczna konwersja typów tylko w miejscach gdzie nie ma ryzyka utraty dokładności
- typy wyliczeniowe umieszczane we odpowiednich zakresach
- obsługa własności, akcesorów (`get`) i mutatorów (`set`)
- mechanizm refleksji typów

C# 1.0 (.NET 1.0, 2002 r.)

- brak konieczności używania wskaźników
- automatyczne zarządzanie pamięcią
- formalne konstrukty klas, interfejsów, struktur, typów wyliczeniowych i delagatów
- możliwość łatwego przeciążania operatorów

C# 2.0 (.NET 2.0, 2005 r.)

- typy generyczne
- metody anonimowe

```
class MyClass<T>
{
    delegate void MyDelegate(T t);
    public void InvokeMethod(T t)
    {
        MyDelegate del = delegate(T item)
        {
            MessageBox.Show(item.ToString());
        };
        del(t);
    }
}
```

- klasy statyczne (static) i częściowe (partial)
- typy nullable (int?...)
- pełna obsługa architektury 64-bitowej

C# 3.0 (.NET 3.5, 2007 r.)

- inicjalizatory obiektów i kolekcji
- LINQ

```
List<int> list = new List<int>() {10, 150, 5, -2, 3};  
IEnumerable matchingItems =  
    from i in list where i >= 10 select i;
```

- metody lambda

```
list.FindAll(val => val >= 10);
```

C# 3.0 (.NET 3.5, 2007 r.)

- typy anonimowe

```
new { Id = 1, Price = 100 }
```

- metody rozszerzające

```
public static void MyMethod(this MyType item);  
  
MyType item = new MyType();  
item.MyMethod();
```

C# 4.0 (.NET 4.0, 2009-2010 r.)

- dynamicznie typowane obiekty (dynamic)

```
int GetLength(dynamic obj)
{
    return obj.Length;
}

// ok, string ma własność Length
GetLength("Hello World");

// ok, tablica ma własność Length
GetLength(new int[] {1, 2, 3});

// rzuca wyjątek, int nie ma własności Length
GetLength(123);
```

C# 4.0 (.NET 4.0, 2009-2010 r.)

- parametry opcjonalne i nazwane

```
public StreamReader OpenTextFile(  
    string path,  
    Encoding encoding = null,  
    bool detectEncoding = false,  
    int bufferSize = 1024  
) { ... }  
OpenTextFile("f.txt", Encoding.UTF8, bufferSize: 4098);
```

- poprawiona współpraca z COM

```
void Increment(ref int x);  
  
// stare podejście  
int foo = 123;  
Increment(ref foo);  
  
// nowe podejście  
Increment(123);
```

Języki platformy .NET

Aktualnie:

- C#
- Visual Basic .NET
- J# (wycofany)
- C++/CLI
- JScript.NET (przestarzały na rzecz Managed JScript)
- Windows PowerShell

.NET 4.0:

- F#
- IronPython
- IronRuby
- M (Oslo)

Języki platformy .NET

Inne języki:

- A# (Ada)
- L#, IronLisp, DotLisp
- P#, Prolog.NET
- S#, #Smalltalk
- Delphi.NET, Oxygene, Component Pascal
- PerlNET, PerlSharp
- Fortran.NET
- NetCOBOL
- AVR.NET (RPG)
- LOLCode.NET
- ...

Wersje .NET

Wersja	Data Wydania	Visual Studio	Windows
1.0	13.02.2002	Visual Studio .NET	
1.1	24.04.2003	Visual Studio .NET 2003	Windows Server 2003
2.0	07.11.2005	Visual Studio 2005	
3.0	06.11.2006		Windows Vista Windows Server 2008
3.5	19.11.2007	Visual Studio 2008	Windows 7
4.0	2009-2010	Visual Studio 2010	

Visual Studio - Edycje

- Visual Studio Express
- Visual Studio Standard
- Visual Studio Professional
- Visual Studio Tools for Office (VSTO)
- Visual Studio Team System

Visual Studio - Porównanie edycji

Produkt	Rozszerzenia	Narzędzia	Projekty instalak	MSDN	Projekt. klas	Refakto-ryzacja
Express	Nie	podst.	CO*	Express	Nie	podst.
Standard	Tak	Tak	CO*	Tak	Tak	Tak
Professional	Tak	Tak	Tak	Tak	Tak	Tak
Team System	Tak	Tak	Tak	Tak	Tak	Tak

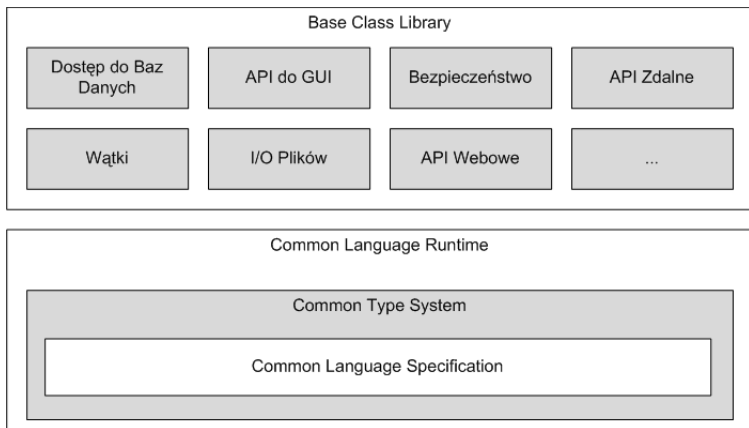
* - ClickOnce

Produkt	Debug	Zdalny debug	Aplikacje 64-bit	Obsługa Itanium	VSTO	Aplikacje mobilne
Express	podst.	Nie	Nie	Nie	Nie	Nie
Standard	Tak	Nie	Tak	Nie	Nie	Tak
Professional	Tak	Tak	Tak	Nie	Tak	Tak
Team System	Tak	Tak	Tak	Tak	Tak	Tak

Mono

- Mono 2.4 - Core API, Visual Basic .NET i C# 2.0
- LINQ do obiektów i XML, Windows Forms 2.0
- Moonlight 1.9 - Silverlight API 1.0
- Moonlight 2.0 Alpha - Silverlight API 2.0
- Mono Develop - "Visual Studio" dla Mono

Struktura .NET



Kod zarządzalny i niezarządzalny

Kod zarządzalny (*managed code*):

Kod uruchamiany na wirtualnej maszynie. Zapewnia programiście większą stabilność oraz bezpieczeństwo.

Kod niezarządzalny (*unmanaged code*):

Kod uruchamiany bezpośrednio na procesorze (serwery COM, aplikacje API Win32).

Pliki binarne .NET i COM/Win32:

Mimo, że nazwy plików binarne .NET mają takie same rozszerzenia jak serwery COM i niezarządzalne pliki binarne Win32 (*.dll lub *.exe) nie mają absolutnie nic wspólnego ze sobą jeśli chodzi o zawartość. W .NET nazywa się je assembly.

Assembly

Nazwa:

- krótka nazwa
- kultura (RFC 1766)
- wersja (`major.minor.build.revision`)
- 64-bitowy hash klucza publicznego użytego do podpisania (*strong name*).

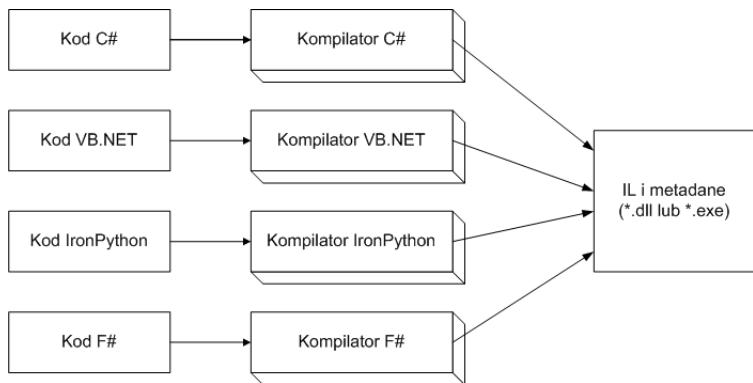
Zawiera:

- kod CIL
- metadane
- manifest

Common Intermediate Language (CIL)

- język pośredni, odpowiednik bytecode'u Javy
- zapewnia integrację języków
- zapewnia niezależność platformy od kodu
- musi być skompilowany w locie przed użyciem
- kompilator JIT (just-in-time), potocznie nazywany Jitter
- osobne kompilatory dla poszczególnych platform (Windows, Pocket PC...) odpowiednio pod nie zoptymalizowane

Kompilacja kodu do CIL



Metadane

- definiują wszystkie typy (klasy, struktury, typy wyliczeniowe...) oraz ich członków (własności, metody, zdarzenia...) używane przez assembly
- tworzone automatycznie przez kompilator

Manifest

- opisuje assembly
- definiuje zewnętrzne assembly, które są niezbędne do prawidłowego działania
- zawiera informacja o wersji, informacje kulturowe, copyright...

Common Type System (CTS)

Common Type System (CTS):

CTS opisuje wszystkie możliwe typy danych obsługiwane przez środowisko uruchomieniowe oraz sposób w jaki te typy mogą wchodzić ze sobą w interakcję. Definiuje również sposób reprezentacji typów w metadanych.

Typy:

- klasy
- interfejsy
- struktury
- typy wyliczeniowe
- delegaty

CTS - Klasy

Klasa:

Klasy zawierają kolekcję własności, metod, zdarzeń, pól i innych członków funkcjonalnie powiązanych ze sobą. W C# definiuje się je słowem kluczowym `class`.

CTS definiuje:

- czy klasa jest zamknięta (`sealed`) czy nie
- czy klasa implementuje jakiś interfejsy
- czy klasa jest abstrakcyjna (`abstract`) czy nie
- jaka jest widoczność klasy (`public`, `internal`...)

CTS - Klasy

```
class MyClass
{
    private int sum = 0;
    public int Sum
    {
        get
        {
            return sum;
        }
        set
        {
            sum += value;
        }
    }
    public void Clear()
    {
        sum = 0;
    }
}
```

CTS - Klasy

```
// klasa abstrakcyjna
abstract class MyClass
{
    ...
}

// klasa zamknięta
sealed class MyClass
{
    ...
}

// klasa implementująca interfejs IEnumerable
class MyClass : IEnumerable
{
    ...
}
```

CTS - Interfejsy

Interfejs:

Interfejsy zawierają nazwaną kolekcję abstrakcyjnych elementów, które muszą być obsługane przez klasę lub strukturę je implementujące. W C# definiuje się je słowem kluczowym `interface`.

W przeciwieństwie do COM, interfejsy nie dziedziczą wspólnego typu w rodzaju `IUnknown`.

```
public interface IMyInterface
{
    void Clear();
}
```

CTS - Struktury

Struktura:

Struktury to "odchudzone" wersje klas, które posiadają semantykę opartą na wartościach. W C# definiuje się je słowem kluczowym `struct`.

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

    public Point(int xPos, int yPos)
    {
        x = xPos; y = yPos;
    }

    public void Display()
    {
        ...
    }
}
```

CTS - Typy wyliczeniowe

Typy wyliczeniowe:

Typy ze ściśle zdefiniowaną listą wartości jakie mogą zostać przez ten typ przyjęte. W C# definiuje się je słowem kluczowym enum.

```
enum CertificateStatus
{
    Active,
    Pending,
    Canceled,
    Revoked
}
```

CTS - Delegaty

Delegaty:

Delegaty są .NETową wersją wskaźników na funkcje. W C# definiuje się je słowem kluczowym `delegate`.

```
// delegat wskazuje na dowolną metodę zwracającą liczbę  
// całkowitą, która jako wejście przyjmuje dwie liczby  
// całkowite  
delegate int BinaryOperation(int x, int y);
```

CTS - Członkowie typów

```
partial class MyClass
{
    // konstruktor
    MyClass()
    {
        myStatus = Status.Active;
    }

    // finalizer (destruktor)
    ~MyClass()
    {
        myStatus = Status.Inactive;
    }

    // statyczny konstruktor
    static MyClass()
    {
        myReadOnlyField = 100;
        myStatus = Status.Active
    }
}
```

CTS - Członkowie typów

```
partial class MyClass
{
    // zagnieżdżony typ
    enum Status
    {
        Active,
        Inactive
    }

    // metoda
    public void MsgBox(string text)
    {
        System.Windows.Forms.MessageBox.Show(text);
    }

    // pole
    public static readonly int myField;

    // pole tylko do odczytu
    private Status myStatus;
}
```

CTS - Członkowie typów

```
partial class MyClass
{
    // własności
    public static int MyProperty
    {
        get { return myField; }
        set { myField = value; }
    }

    protected int MySecondProperty
    {
        get { return myField; }
    }

    private int MyThirdProperty
    {
        get; set;
    }

    const int MYCONST = 15; // stała
}
```

CTS - Członkowie typów

```
partial class MyClass
{
    // indeksy
    private Dictionary<int, string> index;
    public string this[int id]
    {
        get { return index[id]; }
        set { index[id] = value; }
    }

    // zdarzenie
    public event EventHandler OnSomeWeirdAction;

    // operator
    public static MyClass operator +
        (MyClass v1, MyClass v2)
    {
        return new MyClass();
    }
}
```

Common Language Specification (CLS)

Common Language Specification (CLS):

CLS to zbiór reguł, które szczegółowo opisują minimalny i pełny zbiór funkcjonalności, które musi potrafić obsłużyć kompilator, aby wyprodukować kod, który będzie mógł być w pełni dostępny przez wszystkie języki platformy. CLS jest podzbiorem definicji CTS, który jest obsługiwany przez wszystkie języki współpracujące z .NET.

Jeśli w bibliotece wykorzystywane będą jedynie typy z CLS, wszystkie inne języki platformy będą w stanie współpracować z tą biblioteką. Jeśli wykorzystane zostaną typy spoza CLS może to nie być możliwe.

Zasady CLS dotyczą wyłącznie tych części typów, które są dostępne spoza danego assembly.

CLS - Definicje

CLS definiuje między innymi:

- reprezentację ciągów znaków
- wewnętrzną reprezentację typów wyliczeniowych
- sposób definicji statycznych elementów

Sprawdzanie zgodności z CLS:

```
[assembly: System.CLSCompliant(true)]
```

Common Language Runtime (CLR):

Środowisko uruchomieniowe:

Jest to zbiór zewnętrznych usług koniecznych do uruchomienia danej, skompilowanej porcji kodu.

Common Language Runtime (CLR):

Wartwa uruchomieniowa .NET. Jej głównym zadaniem jest lokalizowanie, ładowanie i zarządzanie typami. Oprócz tego odpowiada między innymi za zarządzanie pamięcią, tworzenie domen, wątków i zakresów użycia obiektów, przeprowadzanie różnych testów bezpieczeństwa.

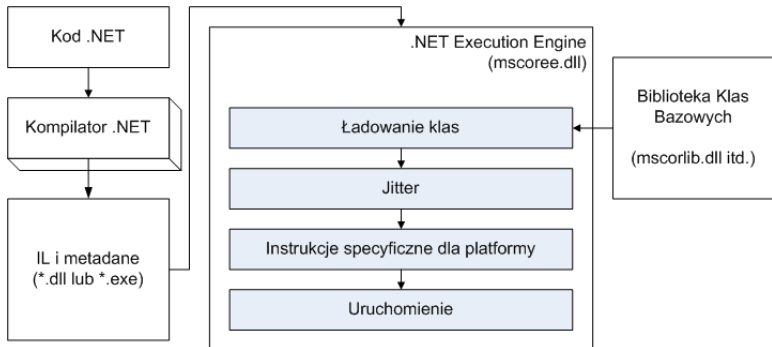
CLR - mscorlib.dll

Sercem CLR jest Common Object Runtime Execution Engine (`mscorlib.dll`). Moduł ten jest ładowany automatycznie.

Zadania `mscorlib.dll`:

- odnajdywanie lokacji assembly oraz odpowiednich typów poprzez czytanie metadanych
- umieszczanie typów w pamięci
- kompilowanie kodu CIL do instrukcji specyficznych dla danej platformy
- testy bezpieczeństwa
- uruchomienie kodu

Workflow CLR



Garbage Collection

Garbage collection:

Architektura zarządzania pamięcią, w której proces zwalniania nieużywanych jej obszarów odbywa się automatycznie.

Wybrane metody:

- reference counting - system plików Unix, Perl, Python
- mark and sweep - Java, C#
- mark and compact - C#
- kopiowanie - Java

Zarezerwowane słowa

C# 1.0

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	finally
fixed	float	for	foreach	goto	if
implicit	in	int	interface	internal	is
lock	long	namespace	new	null	object
operator	out	override	params	private	protected
public	readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string	struct
switch	this	throw	try	typeof	unit
ulong	unchecked	unsafe	ushort	using	virtual
void	volatile	while	false	true	

C# 2.0

yield

C# 3.0

by select	descending var	from where	group	into	orderby
--------------	-------------------	---------------	-------	------	---------

Dyrektywy preprocesora

- C# nie ma osobnego preprocesora, ale poniższe dyrektywy są obsługiwane
- umożliwia definiowania symboli (`#define`, `#undef`), brak możliwości definiowania makr
- kompilacja warunkowa (`#if`, `#elif`, `#else`, `#endif`)
- zwijanie kodu (`#region`, `#endregion`)
- generowanie błędów i ostrzeżeń (`#error`, `#warning`, `#line`)

Dokumentacja XML

```
/// <summary>opis metody</summary>  
/// <param name="filePath">opis parametru filePath</param>  
/// <returns>opis zwracanej wartosci</returns>  
public int LoadXmlFromFile(string filePath)  
{  
    ...  
}
```

Dokumentacja XML - Główne tagi

- `<remarks>...</remarks>`
- `<summary>...</summary>`
- `<example>...</example>`
- `<exception cref="nazwaWyjatku">...</exception>`
- `<param name="nazwaParametru">...</param>`
- `<permission cref="nazwaUprawnien">...</permission>`
- `<returns>...</returns>`
- `<seealso cref="nazwaLinku" />`
- `<include file="nazwaPliku" path="sciezka" />`

Dokumentacja XML - Tagi wspomagające

- `<c>...</c>`
- `<code>...</code>`
- `<list type="bullet|number|table" />`
- `<listheader>...</listheader>`
- `<item>...</item>`
- `<term>...</term>`
- `<description>...<decription>`
- `<para>...</para>`
- `<paramref>...</paramref>`
- `<see cref="nazwaLinku" />`
- `<value>...</value>`

Przestrzenie nazw

Przestrzeń nazw:

Jest to zbiór semantycznie powiązanych typów zawartych w assembly. W C# definiuje się je słowem kluczowym `namespace`.

Jedno assembly może zawierać wiele przestrzeni nazw, z których każda może zawierać semantycznie powiązane typy.

Przestrzenie nazw

```
using System;
namespace MyNamespace
{
    public class MyClass
    {
        static void Main()
        {
            System.Diagnostics.Debug.
                WriteLine("App started");
            Console.WriteLine(
                MySecondNamespace.MyClass.MY_CONST);
        }
    }
}
namespace MySecondNamespace
{
    public static class MyClass
    {
        const string MY_CONST = "Hello World!";
    }
}
```

Przegląd przestrzeni nazw .NET

`System`

podstawowe typy, operacje matematyczne, generacja liczb pseudolosowych, zmienne środowiskowe, powszechnie używane wyjątki i atrybuty

`System.Collections`, `System.Collections.Generic`

kontenery danych, interfejsy do budowania własnych kontenerów

`System.Data`, `System.Data.Odbc`, `System.Data.OracleClient`, `System.Data.OleDb`, `System.Data.SqlClient`

interakcja z relacyjnymi bazami danych przy pomocy ADO.NET

Przegląd przestrzeni nazw .NET

`System.IO`, `System.IO.Compression`, `System.IO.Ports`

operacje wejścia/wyjścia, kompresja danych, operacje na portach

`System.Reflection`, `System.Reflection.Emit`

dynamiczne tworzenie typów, wspomaganie znajdowania typów środowiska uruchomieniowego

`System.Runtime.InteropServices`

interakcja z niezarządzalnym kodem (*.dll z C, serwery COM)

`System.Drawing`, `System.Windows.Forms`

budowa GUI

Przegląd przestrzeni nazw .NET

`System.Windows, System.Windows.Controls, System.Windows.Shapes`

reprezentacja Windows Presentation Foundation

`System.Linq, System.Xml.Linq, System.Data.Linq`

typy używane podczas programowania z użyciem API LINQ

`System.Web`

budowa aplikacji internetowych (ASP.NET)

`System.ServiceModel`

budowa aplikacji rozproszonych w ramach Windows
Communication Foundation

Przegląd przestrzeni nazw .NET

`System.Workflow.Runtime, System.Workflow.Activities`

budowa aplikacji z workflowem w ramach Windows Workflow Foundation

`System.Threading`

budowa aplikacji wielowątkowych

`System.Security`

uprawnienia, obsługa urządzeń kryptograficznych

`System.Xml`

interakcja z XMLem

Podstawowe typy

Typ CTS	C#	CLS	Zakres
System.Byte	byte	Tak	0 do 255
System.SByte	sbyte	Nie	-128 do 127
System.Int16	short	Tak	-32,768 do 32,767
System.UInt16	ushort	Nie	0 do 65,535
System.Int32	int	Tak	-2,147,483,648 do 2,147,483,647
System.UInt32	uint	Nie	0 do 4,294,967,295
System.Int64	long	Tak	-9,223,372,036,854,775,808 do 9,223,372,036,854,775,807
System.UInt64	ulong	Nie	0 do 18,446,744,073,709,551,615
System.Single	float	Tak	$\pm 1.5 * 10^{-45}$ do $\pm 3.4 * 10^{38}$
System.Double	double	Tak	$\pm 5 * 10^{-324}$ do $\pm 1.7 * 10^{308}$
System.Object	object	Tak	
System.Char	char	Tak	U+0000 do U+ffff
System.String	string	Tak	ograniczony pamięcią systemową
System.Decimal	decimal	Tak	$\pm 10e^{-28}$ do $\pm 7.9 * 10e^{28}$
System.Boolean	bool	Tak	true lub false

Typy wartościowe i referencyjne

Typy wartościowe (*value types*):

Porównywane są na podstawie wartości danych, które przechowują. Zawsze posiadają domyślną wartość i mogą być zawsze tworzone lub kopiowane. Są umieszczane na stosie.

Typy referencyjne (*reference types*):

Każda instancja jest różna od pozostałych instancji, nawet jeśli dane w niej przechowywane są identyczne. Nie są umieszczane na stosie lecz na stercie, na której pracuje *garbage collector*.

- wszystkie typy dziedziczą po `System.Object` (metody m.in. `ToString()`, `Equals()` i `GetType()`)
- typy wartościowe dziedziczą po `System.ValueType`

Klasy vs Struktury

- struktury to typy wartościowe (trzymane na stosie)
- klasy to typy referencyjne (trzymane na stercie)
- struktury nie obsługują dziedziczenia
- struktury mogą implementować interfejsy
- klasy mogą implementować interfejsy i obsługują dziedziczenie
- struktury są szybsze
- klasy mają lepsze zarządzanie pamięcią

Pakowanie i rozpakowywanie typów

Pakowanie (*boxing*):

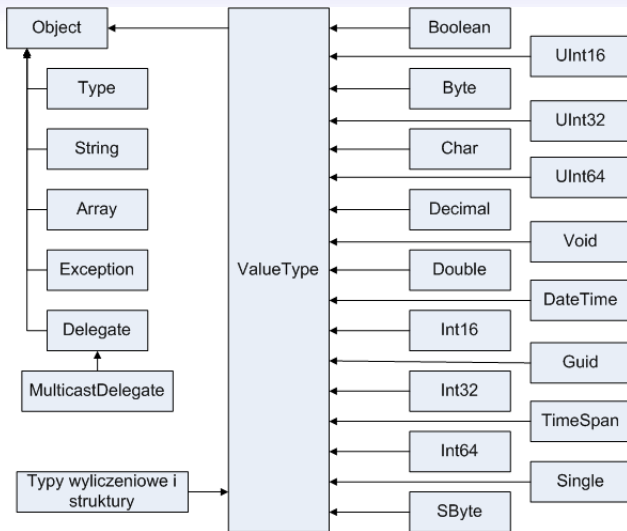
Jest to operacja zamieniająca wartość typu wartościowego na wartość odpowiadającego typu referencyjnego. Pakowanie w C# jest automatyczne.

Rozpakowywanie (*unboxing*):

Jest to operacja zamieniająca wartość typu referencyjnego (wcześniej zapakowanego) na wartość typu wartościowego. Rozpakowywanie w C# wymaga użycia rzutowania.

```
int foo = 123;           // typ wartościowy
object bar = foo;       // pakowanie foo do bar
int foo2 = (int)bar;     // rozpakowanie bar do foo2
```

Hierarchia typów danych



Pętle

- for

```
for (int i = 0; i < 10; i++)  
{  
    Console.WriteLine("Number_□{0}", i);  
}
```

- foreach/in

```
int[] numbers = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
foreach (int number in numbers)  
{  
    Console.WriteLine("Number_□{0}", number);  
}
```

Pętle

- while

```
int i = 0;
while (i < 10)
{
    Console.WriteLine("Number_{0}", i);
    i++;
}
```

- do/while

```
int i = 0;
do
{
    Console.WriteLine("Number_{0}", i);
    i++;
}
while (i < 10); // uwaga na średnik
```

Instrukcje warunkowe

- `if/else if/else`

```
if (myVar == 0)
{
    ...
}
else if (myVar == 1)
{
    ...
}
else
{
    ...
}
```

Instrukcje warunkowe

- switch/case/default

```
switch (myVar)
{
    case 0:
        ...
        break;
    case 1:
        ...
        goto 2; // jawne przenikanie
    case 2:
        ...
        break;
    default:
        ...
        break;
}
```

Źródła

- Pro C# 2008 and the .NET 3.5 Platform Fourth Edition, *Andrew Troelsen*
- MSDN (<http://msdn.microsoft.com>)

Pytania

Pytania?