Platforma .NET – Wykład 2
Składowe platformy .NET

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: 23:02

Spis treści

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

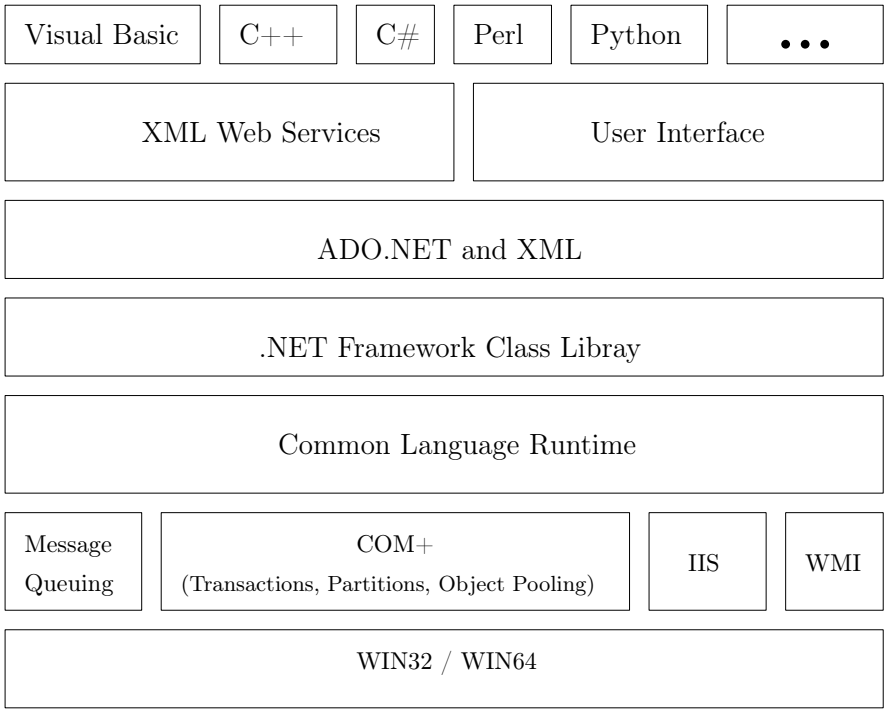
(8) Język zapytań LINQ, Entity Framework
(9) Obsługa standardu XML
(10) Technologia ASP.NET 1/2
(11) Technologia ASP.NET 2/2
(12) Model widok i kontroler – Model View Controller
(13) Tworzenie usług sieciowych SOAP i WCF (komunikacja sieciowa)
(14) Wykład monograficzny .NET 1
(15) Wykład monograficzny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

Plan wykładu nr 2

1. Elementy .NET
   1. główne elementy .NET,
   2. reguły CLS,
   3. system typów.
2. Biblioteka klas
   1. podział typów,
   2. typy wbudowane oraz typy w kontekście C#,
   3. podstawowe klasy w przestrzeni **System**.
3. Podzespół (assembly)/Język pośredni
   1. rola podzespołu,
   2. budowa i model,
   3. język pośredni.

Elementy platformy .NET

- Wspólne środowisko uruchomieniowe oraz biblioteka klas,
- Web Services – zestandaryzowana technologia publikacji serwisów WWW,
- SOAP – Simple Object Access Protocol,
- HTTP – protokół stosowany do przesyłania informacji pomiędzy serwisami WWW,
- XML – opis danych.

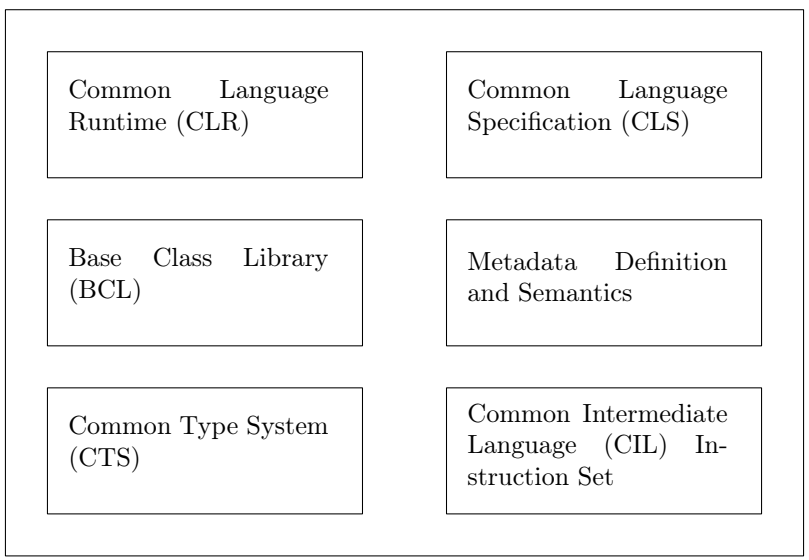


.NET Framework

Oferuje ustalony sposób produkcji oprogramowania w oparciu o platformę w której nie preferuje się określonego (choć C# jest niewątpliwe najbardziej popularny) języka programowania. Platforma zawiera wspólne środowisko uruchomieniowe, bibliotekę klas dodatkowe serwery oraz narzędzia.

Cztery główne filary platformy .NET

- Common Language Runtime (CLR) – środowisko wykonawcze programów .NET odpowiedzialne za lokalizację, wczytanie oraz zarządzanie typami/obiektami platformy .NET
- Common Language Infrastructure (CLI) – ECMA-335
  - Common Language Specification (CLS) – zespół zasad oraz reguł niezbędnych do spełniania aby określony język programowania mógł współpracować z CLI.
  - Common Type System (CTS) – definicja sposobu reprezentacji typów danych w pamięci maszyny.
- Common Intermediate Language (CIL) – język pośredni (IL) stanowiący niezależny zestaw instrukcji od istniejących rozwiązań procesorowych, do którego kompilowane są **wszystkie** programy pracujące w ramach środowiska .NET,
- **metadane** .



Środowisko wykonawcze – maszyna wirtualna

Zadaniem maszyny wirtualnej jest kontrola nad odwołaniami uruchamianego programu do sprzętu i systemu operacyjnego. W przypadku braku pewnych funkcji na poziomie sprzętowym zdaniem maszyny jest symulacja brakującej funkcjonalności. W ten sposób aplikacja/program nie jest wstanie sprawdzić, czy jest uruchomiony na prawdziwym, czy też udawanym/wirtualnym sprzęcie.

Maszyny wirtualne to m.in.:

- interpretery, szczególnie interpretery kodu bajtowego,
- kompilatory JIT,
- emulatory rzeczywiście istniejącego sprzętu, np. emulatory starszych komputerów, konsol do gier.

Uwaga!

Nie zawsze jednoznacznie można określić wprost, czy faktycznie mamy do czynienia z maszyną wirtualną przykładem może być Java.

Zastosowania maszyn wirtualnych

Maszyny wirtualne są używane do różnych celów, min. do:

1. uruchamiania starszych aplikacji i/bądź starszych systemów operacyjnych w innych środowiskach niż pierwotne,
2. zapewnienia uniwersalnego środowiska uruchomieniowego,
3. bezpiecznego rozdzielania zasobów (min. mainframe, superkomputery),
4. uruchamiania jednocześnie różnych systemów operacyjnych na jednym komputerze (bez konieczności „resetu" tzw. systemu goszczącego),
5. tworzenia programowych klastrów,
6. analizowania jego pracy/debuggowania tworzonych systemów operacyjnych,
7. optymalizacji tworzonych aplikacji/programów, poprzez zapewnienie kontroli nad ich środowiskiem działania.

Wymienione właściwości mogą występować łącznie lub być ograniczone, istotną własnością jest zupełność, czyli dana maszyna wirtualna W powinna być w stanie uruchomić swoją własną kopię.

Znane maszyny wirtualne

Kilka współczesnych programowych maszyn wirtualnych:

Nazwa	Model	Zarząd. pam.	Bezpie.	Inter.	JIT C.	CL O.Model	Dynam. typ.
CLR	stack	auto/man	Y	N	Y	Yes	Yes
Mono	stack	auto/man	Y	Y	Y	Yes	Yes
DotGNU	stack	auto/man	Y	N	Y	Yes	N
JVM	stack	auto	Y	Y	Y	Yes	N
LLVM	register	man.	N	Y	Y	Yes	N
Parrot	register	auto	N	Y	N	Yes	Yes

Legenda:

Zarząd. pam. – Zarządzanie pamięcią, Bezpie. – Bezpieczeństwo, Inter. – Interpreter, JIT C. – Just-in-time compiler, CL O.Model – model obiektowy, Dynam. typ. – dynamiczne typowanie.

Maszyny wirtualne można podzielić na dwa podstawowe modele: rejestrowy oraz stosowy. Istotnym problemem jest wydajność maszyny, dlatego obecność kompilatora JIT jest niezbędna, aby dane rozwiązanie oferowało odpowiednio wysoką wydajność.

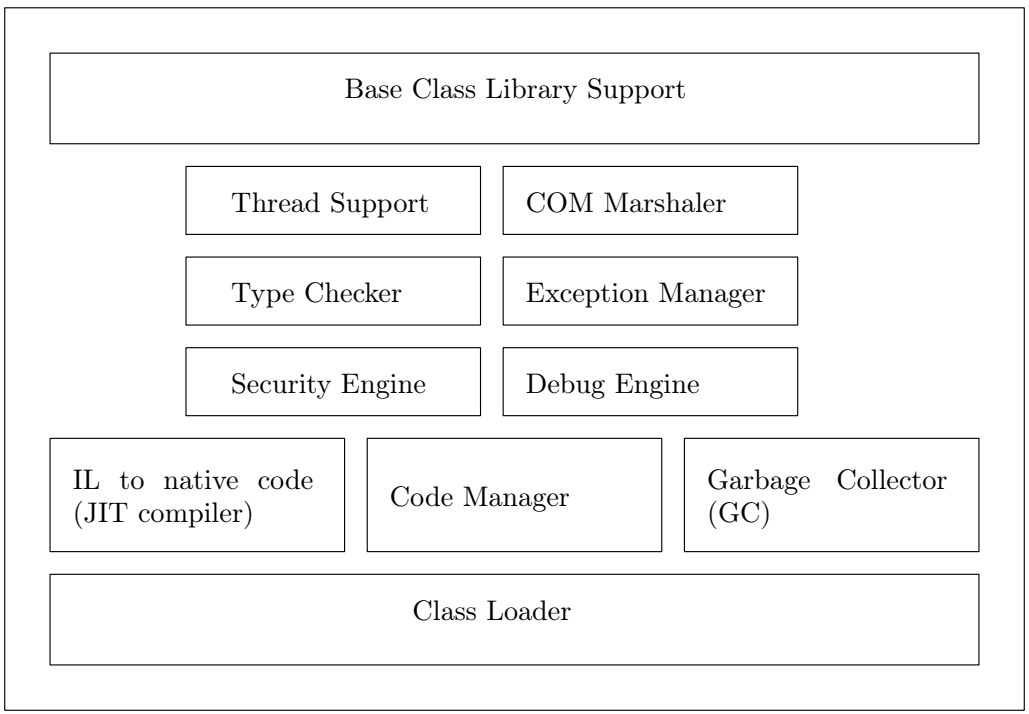
Common Language Runtime (CLR)

Usługi określone przez CLR:

1. zarządzanie kodem, czyli jego uruchamianie oraz nadzór nad jego wykonaniem,
2. zarządzanie pamięcią w przypadku obiektów zarządzanych
3. izolacja obszarów pamięci przydzielonych poszczególnym aplikacją .NET,
4. konwersja języka IL do kodu maszynowego,
5. dostęp do metadanych (informacja o typach),
6. zabezpieczenia w dostępie kodu do zasobów,
7. weryfikacja i zgodność typów,
8. obsługa wyjątków (przekazywanie ich pomiędzy różnymi językami jak np.: F# oraz C#),
9. współpraca kodu zarządzanego z niezarządzanym, obsługa COM i bibliotekami DLL,
10. automatyzacja tworzenia obiektów,
11. usługi wspomagające tworzenie oprogramowania jak „debuggowanie” oraz „profilowanie”

Common Language Runtime (CLR)

Wymagania, co do środowiska wykonawczego zostały określone w dokumencie ECMA-335: `http://www.ecma-international.org/publications/standards/ecma-335.htm`.



Istniejące implementacje wspólnego środowiska wykonawczego:

- CLR – implementacja Microsoftu,
- MONO – `http://www.mono-project.com`,
- DotGNU/Portable.NET – `www.gnu.org/projects/dotgnu/`, `http://dotgnu.org/`.

Rozwój CLR

Kolejne wydania platformy .NET naturalnie pociągają za sobą modyfikację CLR:

			WCF				
			WPF				
			WF	EF	Parallel	Task Model	ASP.NET Core 5
	ASP.NET	ASP.NET	WCS	LINQ	DLR	Windows Store	.NET Native
CLR 1.0	CLR 1.1	CLR 2.0	CLR 2.0	CLR 2.0	CLR 4.0	CLR 4.0	CLR 4.0
.NET 1.0	.NET 1.1	.NET 2.0	.NET 3.0	.NET 3.5 (SP1)	.NET 4.0	.NET 4.5	.NET 4.6

oznacza to, iż dokument ECMA-335 jest poddawany modyfikacjom w momencie wydawania nowych wersji platformy .NET, ostatnia pochodzi z listopada 2016r.

Common Language Specification (CLS)

Rola CLS

CLS to zestaw reguł których spełnienia ma zapewnić współpracę pomiędzy różnymi językami programowania w ramach platformy .NET.

Trzy główne obszary/poziomy, gdzie stosowane są reguły CLS:

1. „framework" – poziom bibliotek/technologii/zestawów klas,
2. „consumer" – język programowania lub narzędzie dostarczające dostęp do funkcjonalności platformy,
3. „extender" – język programowania lub narzędzie dostarczające dostęp do funkcjonalności platformy oraz rozszerza jej możliwości,

Reguły szczegółowe CLS z dokumentu ECMA–335

Przykłady reguł ze standardu:

CLS Rule 17

Unmanaged pointer types are not CLS-compliant.
Note:
CLS (consumer) There is no need to support unmanaged pointer types.
CLS (extender) There is no need to provide syntax to define or access unmanaged pointer types.
CLS (framework) Unmanaged pointer types shall not be externally exported.

CLS Rule 23

System.Object is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS- compliant class.

CLS Rule 25

No longer used. [Note: In an earlier version of this standard, this rule stated „The accessibility of a property and of its accessors shall be identical." The removal of this rule allows, for example, public access to a getter while restricting access to the setter. end note]

Reguły szczegółowe CLS z dokumentu ECMA–335

Przykłady reguł ze standardu:

CLS Rule 38

Properties and methods can be overloaded based only on the number and types of their parameters, except the conversion operators named **op_Implicit** and **op_Explicit**, which can also be overloaded based on their return type.
Note:
CLS (consumer) Can assume that only properties and methods are overloaded, and need not support overloading based on return type unless providing special syntax for operator overloading. If return type overloading isn't supported, then the **op_Implicit** and **op_Explicit** can be ignored since the functionality shall be provided in some other way by a CLS-compliant framework. Consumers must first apply the hide-by- name and hide-by-signature-and-name rules (§8.10.4) to avoid any ambiguity.
CLS (extender) Should not permit the authoring of overloads other than those specified here. It is not necessary to support operator overloading at all, hence it is possible to entirely avoid support for overloading on return type.
CLS (framework) Shall not publicly expose overloading except as specified here. Frameworks authors should bear in mind that many programming languages, including object-oriented languages, do not support overloading and will expose overloaded methods or properties through mangled names. Most languages support neither operator overloading nor overloading based on return type, so **op_Implicit** and **op_Explicit** shall always be augmented with some alternative way to gain the same functionality.

Common Type System (CTS) – Wspólny system typów

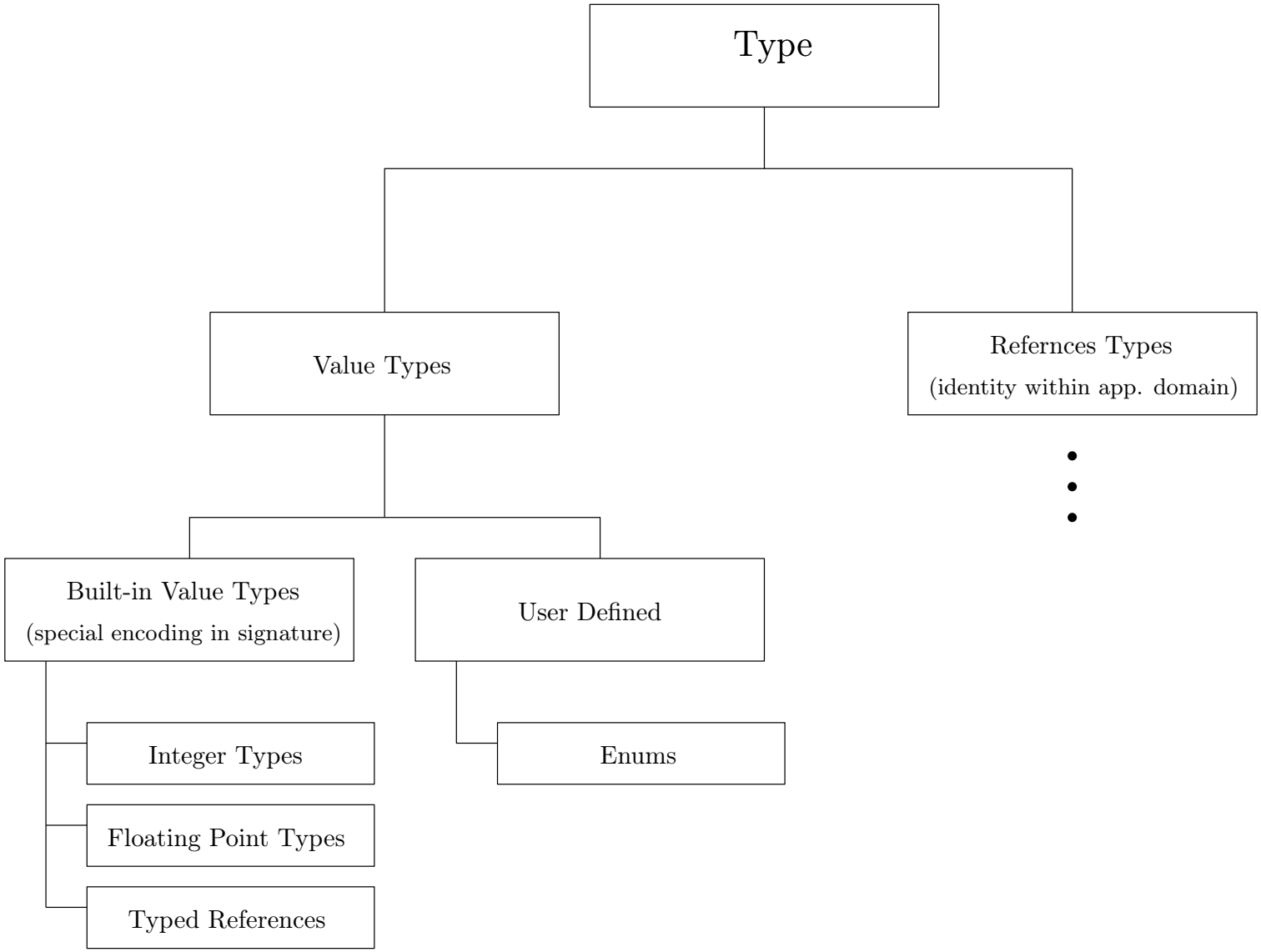
Wspólny system typów określa w jaki sposób typy są deklarowane, używane oraz zarządzane w czasie wykonania (run-time). Najważniejsze funkcje udostępniane przez CTS to min:

1. bezpieczeństwo typów, szybkie wykonywanie kodu, utworzenie niezbędnej struktury do integracji między językowej,
2. pełny model zorientowany obiektowo implementowany w wielu językach programowania .NET,
3. określa warunki, które muszą być spełnione przez określony język .NET, co pozwala na interakcje pomiędzy obiektami pomiędzy różnymi językami .NET .

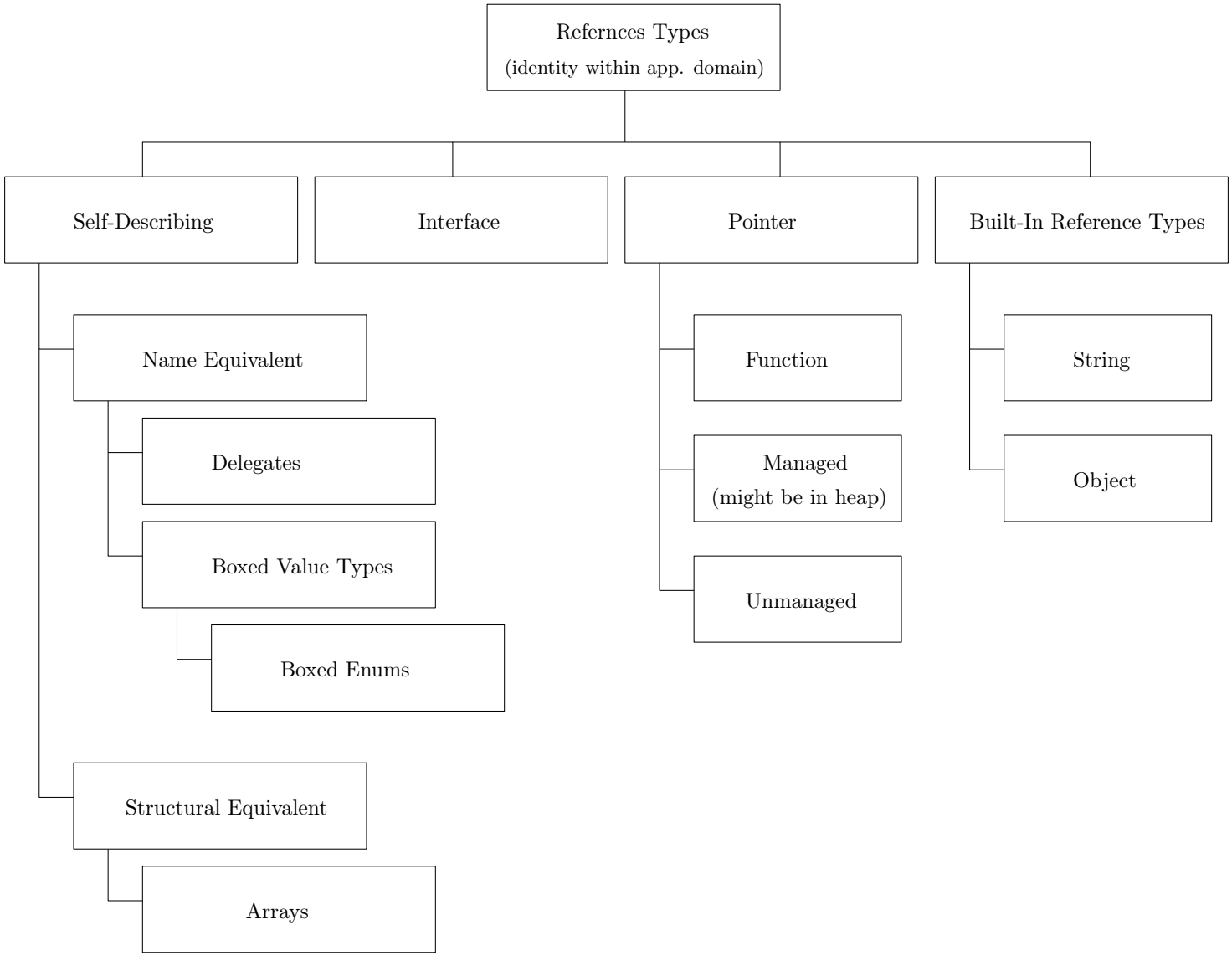
Główne zadanie CTS

Obecność CTS pozwala na integrację języków, bowiem języki programowania obecne w .NET współdzielą te same typy podstawowe określone przez CTS.

Ogólny podział typów



Ogólny podział typów



CTS typy „wbudowane” – built-in

Podstawowe typy obecne we wspólnym środowisku wykonawczym.

Nazwa CIL	Typ CLS	Nazwa BCL	Opis
bool	Yes	System.Boolean	True or false value
char	Yes	System.Char	Unicode 16-bit char
object	Yes	System.Object	Object or boxed value type
string	Yes	System.String	Unicode string
float32	Yes	System.Single	IEC 60559:1989 32-bit float
float64	Yes	System.Double	IEC 60559:1989 64-bit float
int8	No	System.SByte	Signed 8-bit integer
int16	Yes	System.Int16	Signed 16-bit integer
int32	Yes	System.Int32	Signed 32-bit integer
int64	Yes	System.Int64	Signed 64-bit integer
native int	Yes	System.IntPtr	Signed integer, native size
native unsigned int	No	System.UIntPtr	Unsigned integer, native size
typedref	No	System.TypedReference	Pointer plus exact type
unsigned int8	Yes	System.Byte	Unsigned 8-bit integer
unsigned int16	No	System.UInt16	Unsigned 16-bit integer
unsigned int32	No	System.UInt32	Unsigned 32-bit integer
unsigned int64	No	System.UInt64	Unsigned 64-bit integer

CTS a typy w języku C#

(I) Typy wartościowe:
  (1) Proste typy numeryczne i znakowe:
    (a) wielkość integer ze znakiem: sbyte, short, int, long,
    (b) wielkość integer bez znaku: byte, ushort, uint, ulong,
    (c) znak unicode: char,
    (d) IEEE floating point: float, double,
    (e) liczby o dużej precyzji: decimal,
    (f) boolean: bool.
  (2) Typ enum: enum EName { ... },
  (3) Typ struct: struct SName { ... },
  (4) Typy „nullable": typy mają możliwość przyjęcia wartości null.

(II) Typy referencyjne:
  (1) Typ klasowy:
    (a) typ podstawowy dla typu obiektowego: object,
    (b) łańcuchy znaków unicode: string,
    (c) klasy zdefiniowane przez użytkownika: class CName { ... },
  (2) Typ interface: interface IName { ... },
  (3) Typ tablicowy: jedno i wielowymiarowe tablice zdefiniowane przez użytkonika: int[], int[,],
  (4) Typ delegate : typy użytkownika: delegate int DName(...).

Opracowano ponad 40 języków dla .NET

Common CLI Languages

- **A#**: CLI implementation of Ada.
- **Boo**: A statically typed CLI language, inspired by Python.
- **C#**: Most widely used CLI language, bearing similarities to Java, Delphi and C++. Implementations provided by .NET Framework, Portable.NET and Mono.
- **C++/CLI**: A version of C++ including extensions for using CLR objects. Implementation provided only by .NET Framework. Can produce either CIL-based managed code or mixed-mode code that mixes both managed code as well as native code. The compiler is provided by Microsoft.
- **Cobra**: A CLI language with both static as well as dynamic typing, design-by-contract and built-in unit testing.
- **Component Pascal**: A CLI-compliant Oberon dialect. It is a strongly typed language in the heritage of Pascal and Modula-2 but with powerful object-oriented extensions.
- **F#**: A multi-paradigm CLI language supporting functional programming as well as imperative object-oriented programming disciplines. Variant of ML and is largely compatible with OCaml. The compiler is provided by Microsoft. The implementation provided by Microsoft officially targets both .NET and Mono.
- **IronPython**: An open-source CLI implementation of Python, built on top of the DLR.
- **IronRuby**: An open-source CLI implementation of Ruby, built on top of the DLR.
- **IronLisp**: A CLI implementation of Lisp. Deprecated in favor of IronScheme.
- **J#**: A CLS-compliant implementation of Java. The compiler is provided by Microsoft. Microsoft has announced that J# will be discontinued.
- **JScript .NET**: A CLI implementation of ECMAScript version 3, compatible with JScript. Contains extensions for static typing. Deprecated in favor of Managed JScript.
- **L#**: A CLI implementation of Lisp.
- **Managed Extensions for C++**: A version of C targeting the CLR. Deprecated in favor of C++/CLI.
- **Managed JScript**: A CLI implementation of JScript built on top of the DLR. Conforms to ECMAScript version 3.
- **Nemerle**: A multi-paradigm language similar to C#, OCaml and Lisp.
- **Oxygene**: An Object Pascal-based CLI language.
- **P#**: A CLI implementation of Prolog
- **Phalanger**: An implementation of PHP with extensions for ASP.NET
- **Phrogram**: A custom CLI language for beginners and intermediate users produced by The Phrogram Company
- **PowerBuilder**: Can target CLI since version 11.1.
- **Team Developer**: SQLWindows Application Language (SAL) since Team Developer 6.0.
- **VBx**: A dynamic version of VB.NET built on top of the DLR. See VBScript and VBA as this could be thought of being used like a Managed VBScript (though so far this name has not been applied to this) and could be used to replace VBA as well.
- **VB.NET**: A redesigned, object-oriented dialect of Visual Basic. Implementations provided by .NET Framework and Mono.
- **Windows PowerShell**: An object-oriented command-line shell. PowerShell can dynamically load .NET assemblies that were written in any CLI language. PowerShell itself uses a unique scripting syntax, uses curly-braces, similar to other C-based

Other CLI languages

- **Active Oberon** - a CLI implementation of Oberon
- **APLNext** - a CLI implementation of APL
- **AVR.NET** - a CLI implementation of RPG
- **clojure-clr** - a CLI implementation of Clojure
- **Delphi.NET** - a CLI language implementation of the Delphi language.
- **DotLisp** - a CLI language inspired by Lisp
- **Delta Forth .NET** - a CLI implementation of Forth from Dataman
- **dylan.NET** Launchpad·Gitorious - A language targeting the CLR with close relations to MSIL It currently works on .NET and planned support for Mono on all its supported platforms is being worked on.
- **EiffelEnvision** - a CLI implementation of Eiffel
- **Fantom** - a language compiling to .NET and to the JVM
- **Fortran .NET**: Fortran compiling to .NET
- **Gardens Point Modula-2/CLR** - an implementation of Modula-2 that can target CIL
- **GrGen.NET** - a CLI language for graph rewriting
- **IoNET** - a CLI implementation of Io
- **IronScheme** - a R6RS-compliant Scheme implementation built on top of the DLR
- **Ja.NET** - an open source implementation of a Java 5 JDK (Java development tools and runtime) for .NET
- **Common Larceny** - a CLI implementation of Scheme
- **LOLCode.NET** - a CLI implementation of LOLCODE
- **Mercury on .NET** - an implementation of Mercury that can target CIL
- **Net Express** - a CLI implementation of COBOL
- **NetCOBOL** - a CLI implementation of COBOL
- **COBOL2002 for .NET Framework** - a CLI implementation of COBOL
- **COBOL2002 for .NET Framework** - a CLI implementation of COBOL
- **OxygenScheme** - a CLI implementation of Scheme
- **PL/IL** - a CLI implementation of PL/I
- **#S** - A CLI language that implements Scheme (a port of Peter Norvig's Jscheme).
- **#Smalltalk** - a CLI implementation of Smalltalk
- **sml.net** - a CLI implementation of Standard ML
- **Synergy.NET** - a CLI implementation of DIBOL
- **Visual COBOL** - a CLI implementation of COBOL
- **X#** - a CLI implementation of ASM developed for Cosmos. X# was also the codename for the XML-capabilities of Cω.
- **Zonnon**, Yet another CLI-compliant Oberon dialect.

Najważniejsze języki

Najważniejsze języki dostępne „z pudełka” to:

- C#, Visual Basic .NET, C++/CLI,
- J# (odmiana języka Java opracowany przez Microsoft), JScript .NET (kompilowana odmiana języka JScript).

Inne ważne języki:

1. COBOL, Delphi.NET, Oxygen,
2. Eiffel, Fortran, Lisp,
3. Nemerle, F#, Python, P# (Prolog).

F# – język funkcyjny

Definicja funkcji rekurencyjnej **fib** obliczającej liczby ciągu Fibbonaciego:

```
let rec fib x =
    match x with
        | 1 -> 1
        | 2 -> 1
        | x -> fib (x - 1) + fib (x - 2)
printfn "(fib 2) = %i" (fib 2)
printfn "(fib 6) = %i" (fib 6)
printfn "(fib 11) = %i" (fib 11)
```

i elementy ciągu Lucasa:

```
let rec luc x =
    match x with
        | x when x <= 0 -> failwith "value must be greater than 0"
        | 1 -> 1
        | 2 -> 3
        | x -> luc (x - 1) + luc (--x - 2)
printfn "(luc 2) = %i" (luc 2)
printfn "(luc 6) = %i" (luc 6)
printfn "(luc 11) = %i" (luc 11)
printfn "(luc 12) = %i" (luc 12)
```

Konwersja liczby całkowitej na postać binarną:

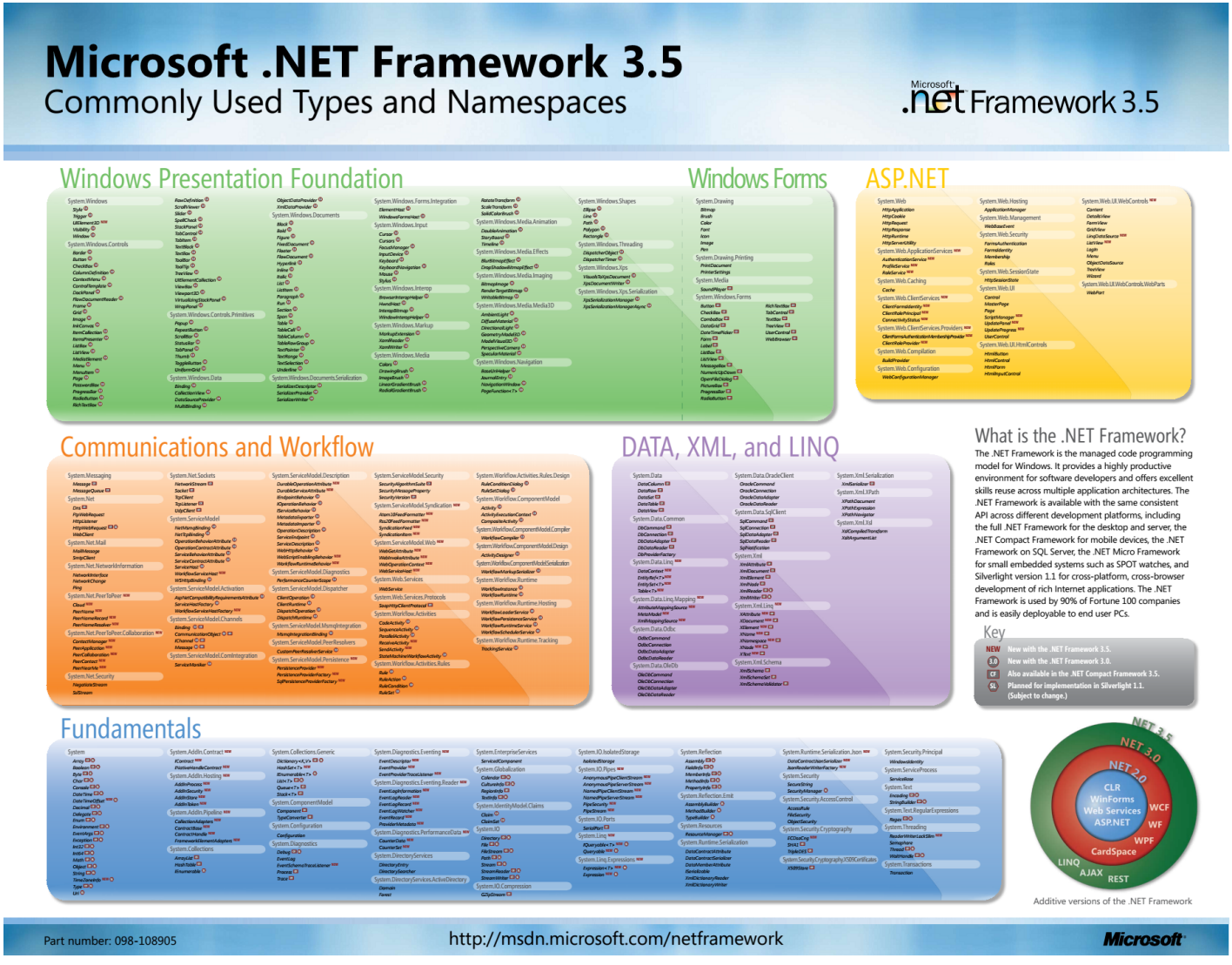
```
let BitsNum = 32 ;;

let binary_of_int n =
    [ for  i in BitsNum -  1  .. -1 .. 0 ->
        if (n  >>>  i) %  2  =  0  then "0"  else "1"  ]
    |>  String.concat "" ;;

let x1 = binary_of_int    1431 ;;
let x2 = binary_of_int ( -1123 ) ;;
```

Zadania i funkcje biblioteki klas

Standard ECMA-335 określa także postać standardowej biblioteki klas, zaliczane są następujące pakiety (zachowane zostało oryginalne nazewnictwo standardu):

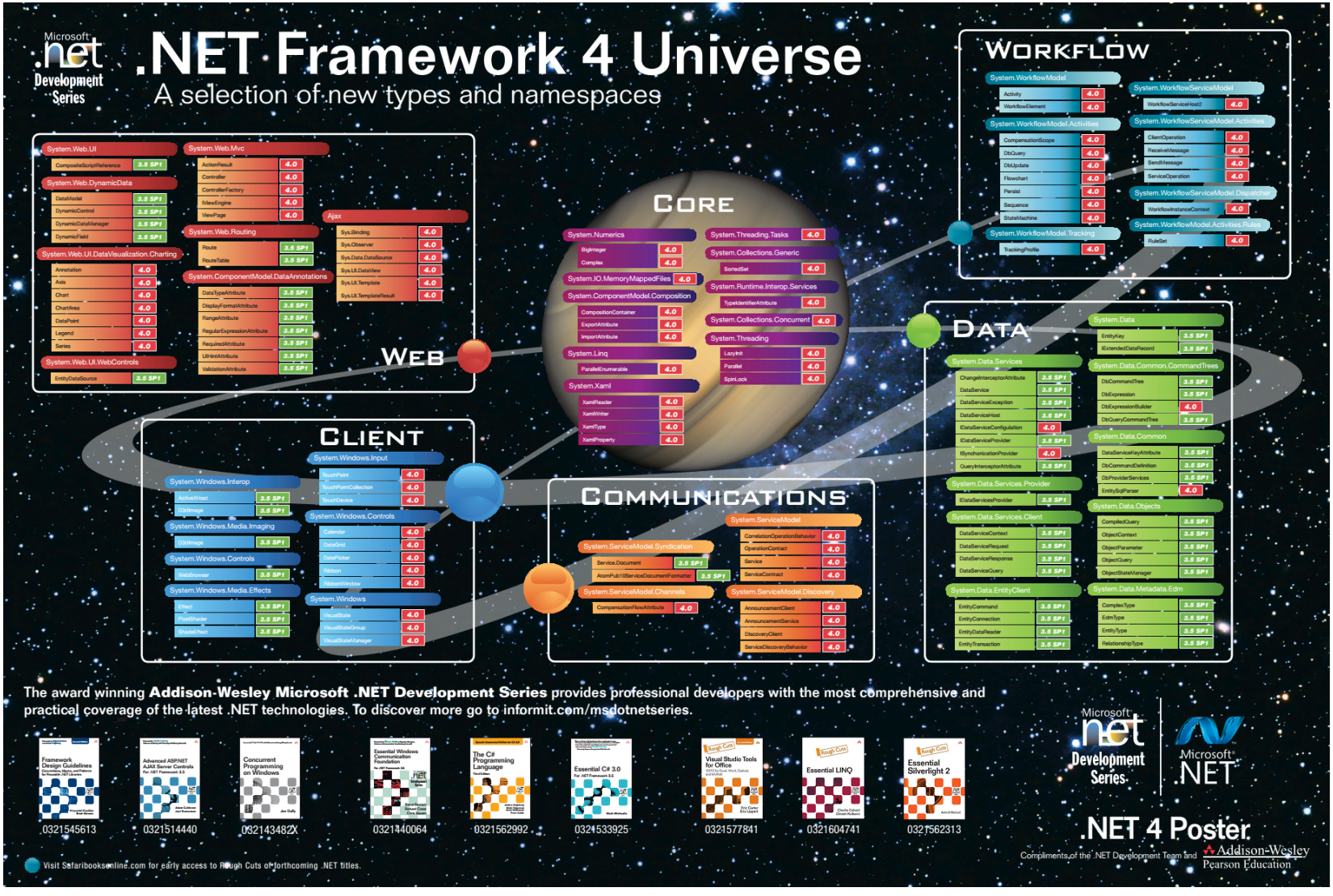
1. Run-time infrastructure library
2. Base Class Library (BCL)
3. Network library
4. Reflection library
5. XML library
6. Extended numerics library
7. Extended array library
8. Vararg library
9. Parallel library

Ogólny schemat Base Class Library – BCL – .NET 3.5



plakat autorstwa firmy Microsoft

Ogólny schemat Base Class Library – BCL – .NET 4.0



plakat autorstwa firmy Addison-Wesley

Kilka klas z przestrzeni **System**

Przestrzeń nazw	Krótki opis
System	Zawiera klasy definiujące najczęściej używane typy danych, zdarzenia i obsługę zdarzeń a także interfejsy, atrybuty oraz wyjątki
System.CodeDom	Reprezentacja dokumentu/kodu źródłowego w postaci drzewa
System.ComponentModel	Tworzenie komponentów, kontrolek, rejestracja oraz adaptacja
System.Collections,	Różne kolekcje obiektów min.: listy, kolejki, tablice bitów, tablice skrótów i słowniki
System.Configuration	Dostęp do ustawień konfiguracyjnych .NET i podzespołów oraz obsługa błędów w plikach konfiguracyjnych (pliki * config)
System.IO	Podstawowy dostęp i zarządzanie strumieniami danych
System.Text	Kodowanie i konwersje znaków, manipulacje łańcuchami znaków
System.TextRegular	Obsługa wyrażeń regularnych

Assemblies, czyli podzespoły

Assembly/Assemblies (podzespół/zestaw/podzespoły/zestawy)

Podzespół to logiczny blok, skompilowany do kodu pośredniego. Z tych elementów budowane są aplikacje .NET, zawierają one kod w języku IL oraz metadane. Podzespół opatrzony jest numerem wersji i jest on zamkniętą spójną całością udostępniającym określoną funkcjonalność.

Istnieją dwa rodzaje metadanych: w pierwszym podzespół jest traktowany jako jednostka i jest to tzw. manifest oraz metadane przeznaczone do opisu poszczególnych typów zawartych w podzespole.

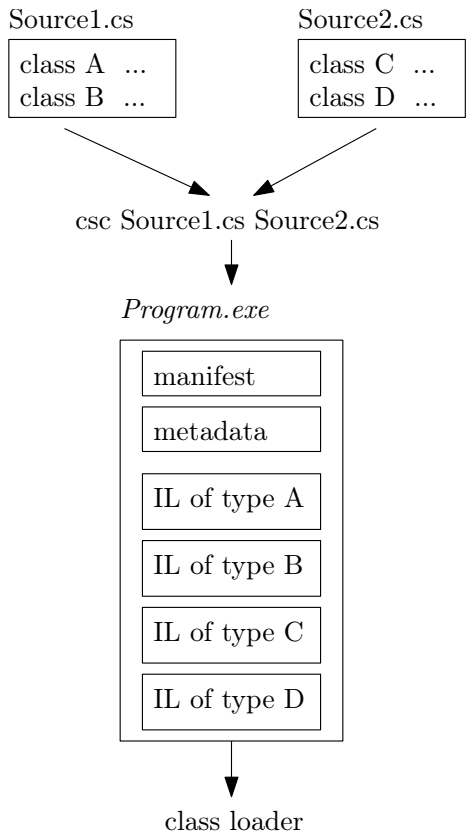
Manifest jest częścią podzespołu i opisuje jego kod i zasoby poprzez podanie:

- nazwy (tzw. tożsamości) podzespołu,
- numer wersji oraz tzw. opisu kultury (informacje regionalne oraz informacje o językach obsługiwanych przez dany podzespół),
- podpisu cyfrowego dla podzespołu,
- pliki zawarte w podzespole,
- typy i zasoby zawarte w podzespole wraz z informacją które typy/zasoby są dostępne do użycia przez inne podzespoły,
- powiązania z innymi podzespołami,
- uprawnienia które są wymagane do poprawnej pracy podzespołu.

Assemblies, czyli podzespoły

Metadane typów zawierają opis typów zdefiniowanych w kodzie zarządzanym. Zawarte są w tym samym pliku co kod IL, zawierają min:

1. Opis typów:
   - nazwa typu,
   - zasięg typu (publiczny bądź ograniczony do podzespołu)
   - nazwę typu po którym typ dziedziczy,
   - interfejsy zaimplementowane,
   - metody zaimplementowane,
   - właściwości typu,
   - zdarzenia obsługiwane przez typ.
2. Opis atrybutów
   - atrybuty posiadają określone nazwy i funkcje określone w środowisku .NET,
   - określają sposób wykonywania kodu bądź opisują wymagania co do bezpieczeństwa.

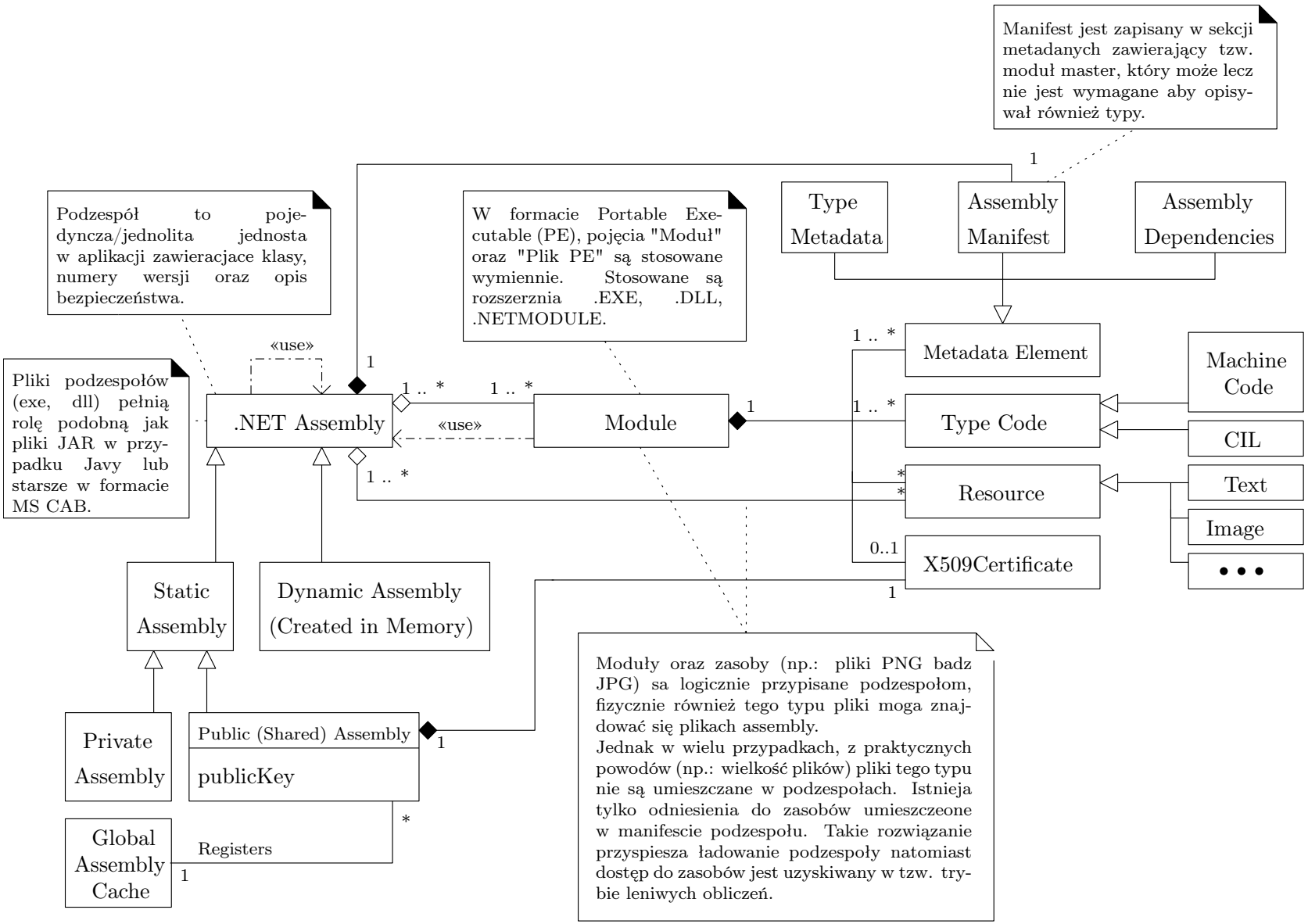


Assemblies, czyli podzespoły

Ważne elementy związane z podzespołami to:

1. po kompilacji do kodu IL, kod zarządzany jest częścią podzespołu składającego się z jednego bądź wielu plików DLL/EXE,
2. podzespoły mogą być określone jako prywatne bądź współdzielony umieszczone są w GAC (Global Assembly Cache),
3. podzespół może zostać podpisany kluczem cyfrowym (sygnaturą) oraz opatrzony
4. informacje o prywatnym podzespole nie są wprowadzane do platformy .NET (rejestr nie jest modyfikowany),
5. podzespoły w katalogu aplikacji nie wymagają dodatkowych operacji,
6. współużytkowanie podzespoły są umieszczene w GAC,
7. podzespół może mieć tzw. silną nazwę (podpis/sygnatura cyfrowa) weryfikowana podczas ładowania podzespołu lub słabą nazwę,
8. wersja podzespołu w postaci G.D.K.K (główny/major, drugorządny/minor, kompilacji/compilation, korekty/revision) np.: 3.0.1240.2.
9. wersja informacyjna, czyli ciąg znaków czytelnych dla ludzi,
10. numery wersji podzespołów od których zależy dany podzespół.

Model podzespołu (assembly model)



Język pośredni / Kod zarządzany

Język pośredni lub wspólny język pośredni (ang. Common Intermediate Language – CIL) to język niskiego poziomu dla platformy .NET. Stanowi on odpowiednik assemblera dla języków wysokiego poziomu jak C/C++. Ogólnie CIL jest podobny do assemblera, jednak został on wyposażony w konstrukcje programowania obiektowego i nie jest wykonywany przez stosową maszynę wirtualną ale kompilowany przez JIT/ngen do kodu maszynowego.
Kod zarządzany, to kod zgodny ze specyfikacją CLR (CLS) zawiera również metadane. W ten sposób umożliwia się realizację: usług automatycznego zarządzania zasobami, współpracy fragmentów/podzespołów oprogramowania napisanych w różnych językach programowania, zarządzania uprawnieniami w dostępie do podzespołu oraz ułatwia zarządzanie cyklem życia obiektów (min. system odśmiecający – garbage collector).

MSIL czy IL

Początkowo język pośredni nazywał Microsoft Intermediate Language (MSIL), jednak po procedurze standaryzacji C# oraz CLI obowiązująca nazwa to CIL lub IL.

Kompilacja i wykonywanie programów IL

Ogólne podejście do kompilacji:

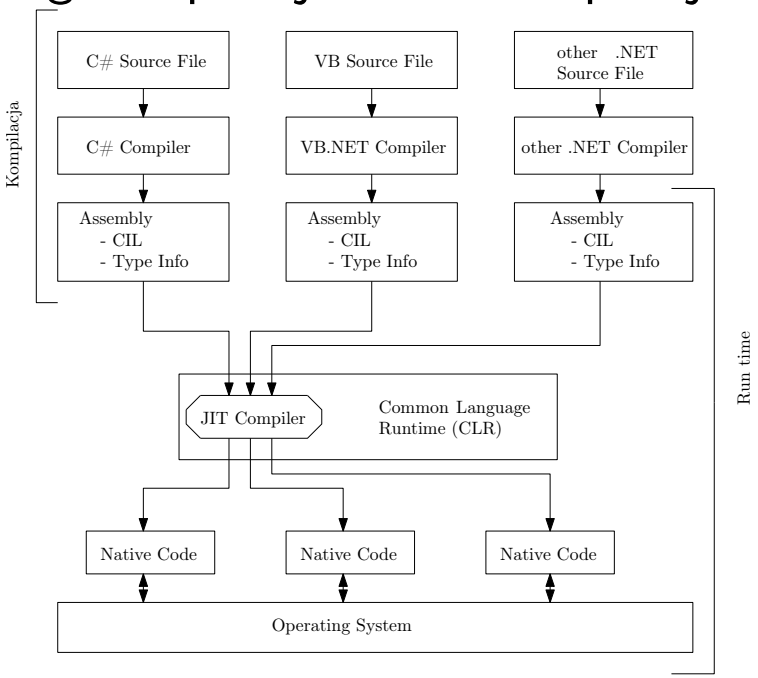
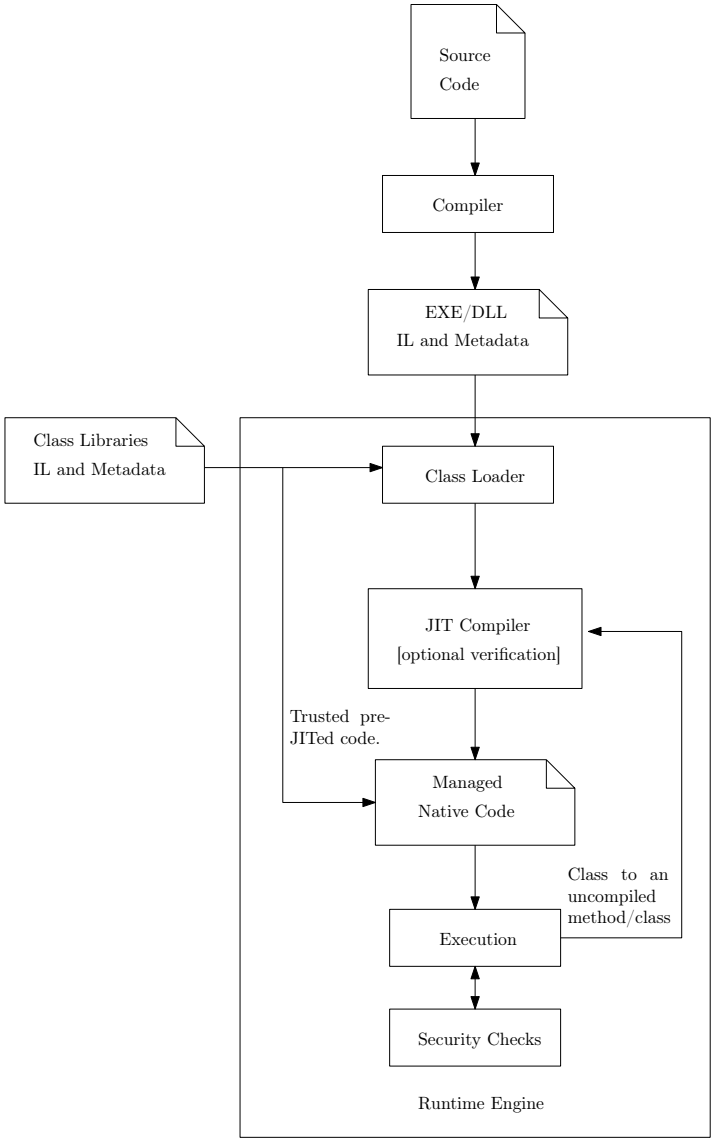


Diagram kompilacji i wykonania:



Podział instrukcji IL

Instrukcje kodu bajtowego CIL można podzielić na następujące grupy:

1. Wczytaj i Zapisz – Load and store
2. Arytmetyczne – Arithmetic
3. Konwersji typów – Type conversion
4. Tworzenia i manipulacji obiektami – Object creation and manipulation
5. Instrukcje zarządzania stosem – Operand stack management (push / pop)
6. Instrukcje sterujące – Control transfer (branching)
7. Wywołania metody i powrotu – Method invocation and return
8. Zgłaszania sytuacji wyjątkowej – Throwing exceptions
9. Realizujące współbieżność – Monitor-based concurrency

Przykład – pusty program

Kod w IL, nie wykonuje żadnych istotnych czynności:

```
.assembly hello1 { }
.class hello1 {
    .method static public void main() il managed  {
        .entrypoint
        ret
    }
}
```

Instrukcje **.entrypoint** a **ret** stanowią „w pewnym sensie" odpowiednik funkcji main.

Przykład – wyświetlenie komunikatu

Kod w IL, wyświetla komunikat tekstowy:

```
.assembly hello2 { }
.class hello2 {
    .method static public void main() il managed {
        .entrypoint
        ldstr "Witojcie!!!"
        call void [mscorlib]System.Console::WriteLine(class System.String)
        ret
    }
}
```

Można powiedzieć, że assembler IL to assembler obiektowy.

Odczytanie liczby całkowitej i wyświetlenie jej na ekranie

```
.assembly hello3 { }
.method public static void Main() il managed {
    .entrypoint
    .maxstack 2
    .locals (int32 V_0)
    ldstr "Liczba calkowita int32:"
    call void [mscorlib]System.Console::WriteLine(string)
    call string [mscorlib]System.Console::ReadLine()
    call int32 [mscorlib]System.Int32::Parse(string)
    stloc.0
    ldstr "Wpisano liczbe {0}."
    ldloc.0
    box [mscorlib]System.Int32
    call void [mscorlib]System.Console::WriteLine(string, object)
    ret
}
```

Jednak bez klas też można tworzyć programy w IL'u.

Porównanie dwóch liczb całkowitych

```
.assembly big_and_small_nums { }
.method public static void Main() il managed {
  .entrypoint
  .maxstack  2
  .locals init (int32 V_0, int32 V_1)
  IL_0000:  ldstr      "Podaj pierwsza liczbe"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  call       string [mscorlib]System.Console::ReadLine()
  IL_000f:  call       int32 [mscorlib]System.Int32::Parse(string)
  IL_0014:  stloc.0
  IL_0015:  ldstr      "I jeszcze druga"
  IL_001a:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_001f:  call       string [mscorlib]System.Console::ReadLine()
  IL_0024:  call       int32 [mscorlib]System.Int32::Parse(string)
  IL_0029:  stloc.1
  IL_002a:  ldloc.0
  IL_002b:  ldloc.1
  IL_002c:  bne.un.s   IL_003a
  IL_002e:  ldstr      "Obie wielkosci sa rowne"
  IL_0033:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0038:  br.s       IL_0054
  IL_003a:  ldloc.0
  IL_003b:  ldloc.1
  IL_003c:  bge.s      IL_004a
  IL_003e:  ldstr      "Druga liczba jest wieksza niz pierwsza"
  IL_0043:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0048:  br.s       IL_0054
  IL_004a:  ldstr      "Druga liczba jest mniejsza niz pierwsza"
  IL_004f:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0054:  ldstr      "Nacisnij Enter aby zakonczyc program"
            call       void [mscorlib]System.Console::WriteLine(string)
            call       string [mscorlib]System.Console::ReadLine()
  IL_0059:  pop
  IL_005a:  ret
}
```

Porównanie dwóch liczb całkowitych

```
using System;
namespace Test1a {
    class Program  {
        public static void Main(string[] args) {
            Console.WriteLine("Podaj pierwsza liczbe");
            int x = Int32.Parse( Console.ReadLine() );
            Console.WriteLine("I jeszcze druga");
            int y = Int32.Parse( Console.ReadLine() );

            if ( x == y ) {
                Console.WriteLine("Obie wielkosci sa rowne");
            }
            else {
                if ( x < y ) {
                    Console.WriteLine("Druga liczba jest wieksza niz pierwsza");
                }
                else {
                    Console.WriteLine("Druga liczba jest mniejsza niz pierwsza");
                }
            }
            Console.WriteLine("Nacisnij Enter aby zakonczyc program");
            Console.ReadLine();
        }
    }
}
```

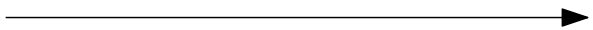
Definicja metod

```
.assembly extern mscorlib {}

.assembly Methods { .ver 1:0:1:0 }
.module Methods.exe

.method static void main() cil managed {
    .maxstack 2
    .entrypoint

    ldc.i4 15
    ldc.i4 27
    call int32 SumCalculate(int32, int32)
    call void WriteNum(int32)
    ret
}

.method public static int32 SumCalculate (int32 , int32 ) cil managed {
    .maxstack 2

    ldarg.0
    ldarg.1
    add

    ret
}
.method public static void WriteNum(int32) cil managed {
    .maxstack 2
    ldstr "Wartosc argumentu: "
    call void [mscorlib]System.Console::Write(string)

    ldarg.0
    call void [mscorlib]System.Console::Write(int32)

    ret
}
```

Od C# przez IL do assemblera Intel x86/x64

Source code in C#

```
Vector2D v;
v = new Vector();
v.x = 10;
v.y = 20;
```



IL source

```
.locals init ([0] class Vector2D v)
newobj instance void Vector2D::.ctor()
stloc.0

ldloc.0
ldc.i4 0x0A
stfld int32 Vector2D::x

ldloc.0
ldc.i4 0x14
stfld int32 Vector2D::y
```

Source code in Intel x86_32 assembly

```
call ED00AAD1h
mov ecx, eax
call dword ptr ds:[00Af0012h]
mov dword ptr [ecx+4], 000Ah
mov dword ptr [ecx+8], 0014h
```

Tabela instrukcji – wybór

Opcode	Instrukcja	Opis
0x58	add	Add two values, returning a new value
0xD6	add.ovf	Add signed integer values with overflow check
0xD7	add.ovf.un	Add unsigned integer values with overflow check
0x3E	ble <int32 (target)>	Branch to target if less than or equal to
0x8C	box <typeTok>	Convert a boxable value to its boxed form
0x38	br <int32 (target)>	skok do etykiety target
0x2B	br.s <int8 (target)>	j.w. krótka forma
0x01	break	Inform a debugger that a breakpoint has been reached
0x28	call <method>	Call method described by method
0xFE 0x02	cgt	Push 1 (of type int32) if value1 > value2, else push 0
0xC3	ckfinite	Throw ArithmeticException if value is not a finite number
0x73	newobj <ctor>	Allocate an uninitialized object or value type and call ctor
0x04	ldarg.2	Load argument 2 onto the stack
0x11	ldloc.s <uint8 (indx)>	Load local variable of index indx onto stack, short form
0xFE 0x0D	ldloca <uint16 (indx)>	Load address of local variable with index indx
0x62	shl	Shift an integer left (shifting in zeros), return an integer
0x63	shr	Shift an integer right (shift in sign), return an integer
0x7A	throw	Zgłoszenie wyjątku
0x79	unbox <valuetype>	Extract a value-type from obj, its boxed representation
0xFE 0x13	volatile. [prefix]	Subsequent pointer reference is volatile
0x61	xor	Bitwise XOR of integer values, returns an integer

Język pośredni – podsumowanie

Czy warto poznawać assembler IL?

Tak, choć naturalnie tworzenie dużych programów raczej mija się z celem, jednak poznanie zagadnień związanych z „IL - asm" na pewno przyczyni się do pogłębienia wiedzy o wewnętrznych mechanizmach CLR oraz CLI. Co jest szczególne ważne, jeśli planuje się tworzenie własnego kompilatora do platformy .NET.

W następnym tygodniu między innymi

Wykład „Programowanie w C# – środowisko Visual Studio 2010, MonoDevelop, syntaktyka C#, wyjątki, współpraca z DLL”,

1. przegląd narzędzi dla programistów w platformie .NET, min. ildasm, gacutil,
2. podstawowe informacje o C#
3. pojęcie klasy,
4. podstawowe informacje wejścia/wyjścia
5. tablice, ciągi znaków, wyrażenia regularne,
6. współpraca z kodem niezarządzanym,

Proponowane tematy prac pisemnych:

1. analiza i porównanie bibliotek standardowych dla języka Java oraz platformy .NET,
2. różnice pomiędzy IL dla platformy .NET a językiem pośrednim dla maszyny wirtualnej JAVA,
3. opis technicznej struktury oraz roli podzespołów.

Dziękuje za uwagę!!!