

IL Assembler Today and Tomorrow

Serge Lidin (Microsoft)

Agenda

- IL Assembler Technology Overview
 - Unique positioning of IL Assembler
 - Compilers Employing IL Assembler
 - Build Environments Employing IL Assembler
- Recent Developments
 - Generics
 - Custom Attribute encoding
 - Declarative security encoding
 - Compilation control directives
 - Reference aliasing (**.typedef** directive)
 - Targeting 64 bit platforms
 - Type forwarding
 - IL Disassembler new features
- Future Work
 - Componentization
 - Nautilus Project
 - Phoenix Project

IL Assembler Technology Overview

- Unique positioning of IL Assembler
 - Platform-driven paradigm
 - Own language of .NET CLR
 - More than full functionality
 - Popular back-end for compilers and tools

IL Assembler Technology Overview

- **Compilers Employing IL Assembler**
 - Ada# (USAF Academy, Colorado)
 - Alice.NET (Saarland University, Saarbrücken)
 - Boo (codehaus.org)
 - COBOL
 - NetCOBOL (Fujitsu)
 - COBOL2002 for .NET Framework (NEC/Hitachi)
 - NetExpress (Microfocus)
 - CommonLarceny.NET (Northeastern University, Boston)
 - CULE.NET (CULEPlace.com)
 - Component Pascal (Queensland University of Technology, Australia)
 - Fortran (Lahey/Fujitsu)
 - Hotdog Scheme (Northwestern University, Chicago)

IL Assembler Technology Overview

- **Compilers Employing IL Assembler (continued)**
 - Lagoon.NET (University of California, Irvine)
 - LCC (ANSI C) (Microsoft Research, Redmond)
 - Mercury (University of Melbourne, Australia)
 - Modula-2 (Queensland University of Technology, Australia)
 - Moscow ML.NET (Royal Veterinary and Agricultural University, Denmark)
 - Oberon.NET (Swiss Federal Institute of Technology, Zürich)
 - S# (Smallscript.com)
 - SML.NET (Microsoft Research, Cambridge, UK)
 - ...did I miss someone?

IL Assembler Technology Overview

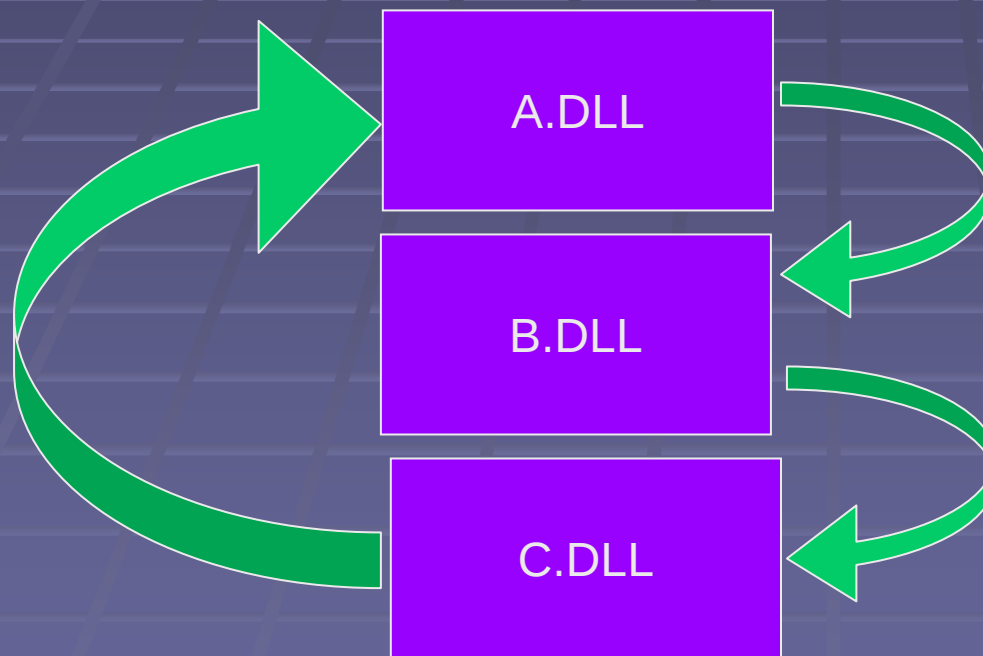
- Build Environments Employing IL Assembler
 - ASMMETA: Resolving Circular Dependencies
 - ILLINK: Linking managed modules
 - Mike Stall's IL inlining in C# and VB
 - Preemptive's DotFuscator

IL Assembler Technology Overview

- Build Environments Employing IL Assembler
 - ASMMETA: Resolving Circular Dependencies
 - ILLINK: Linking managed modules
 - Mike Stall's IL inlining in C# and VB
 - Preemptive's DotFuscator

ASMMETA: Resolving Circular Dependencies

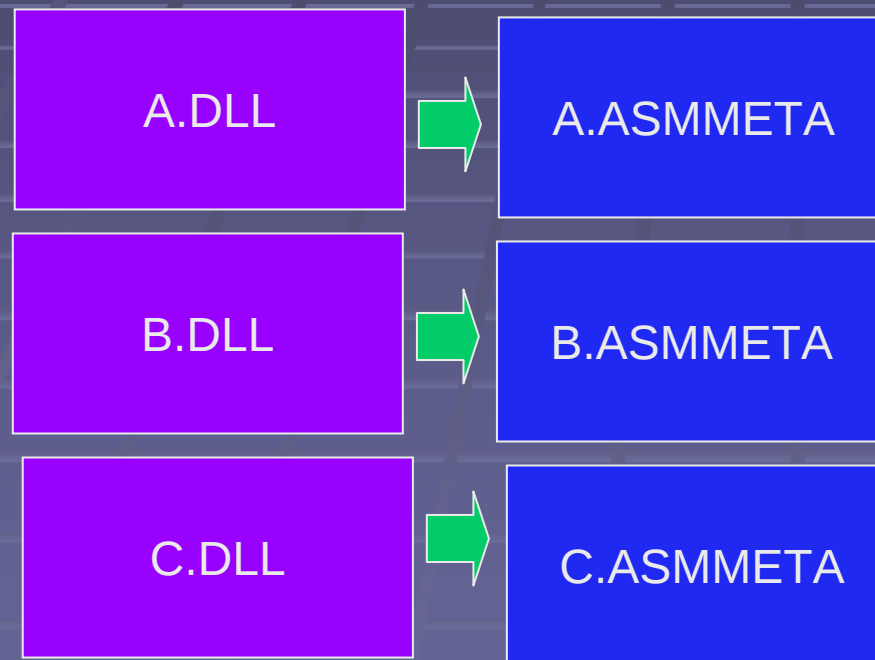
Circular dependency – How to compile?



*C# and VB compilers
require all referenced
assemblies to be present
at compile time.*

ASMMETA: Resolving Circular Dependencies

Step 1: Create .asmmeta files



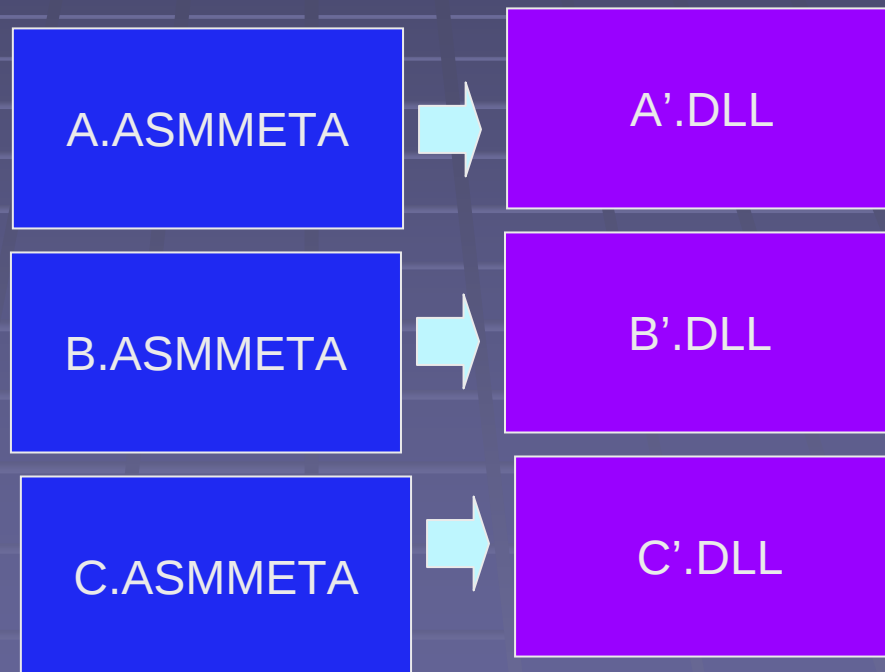
*.asmmeta files are in fact
IL Assembler files without
method bodies and
without private class
members*

ASMMETA: Resolving Circular Dependencies

Step 2: Create “dummy” assemblies

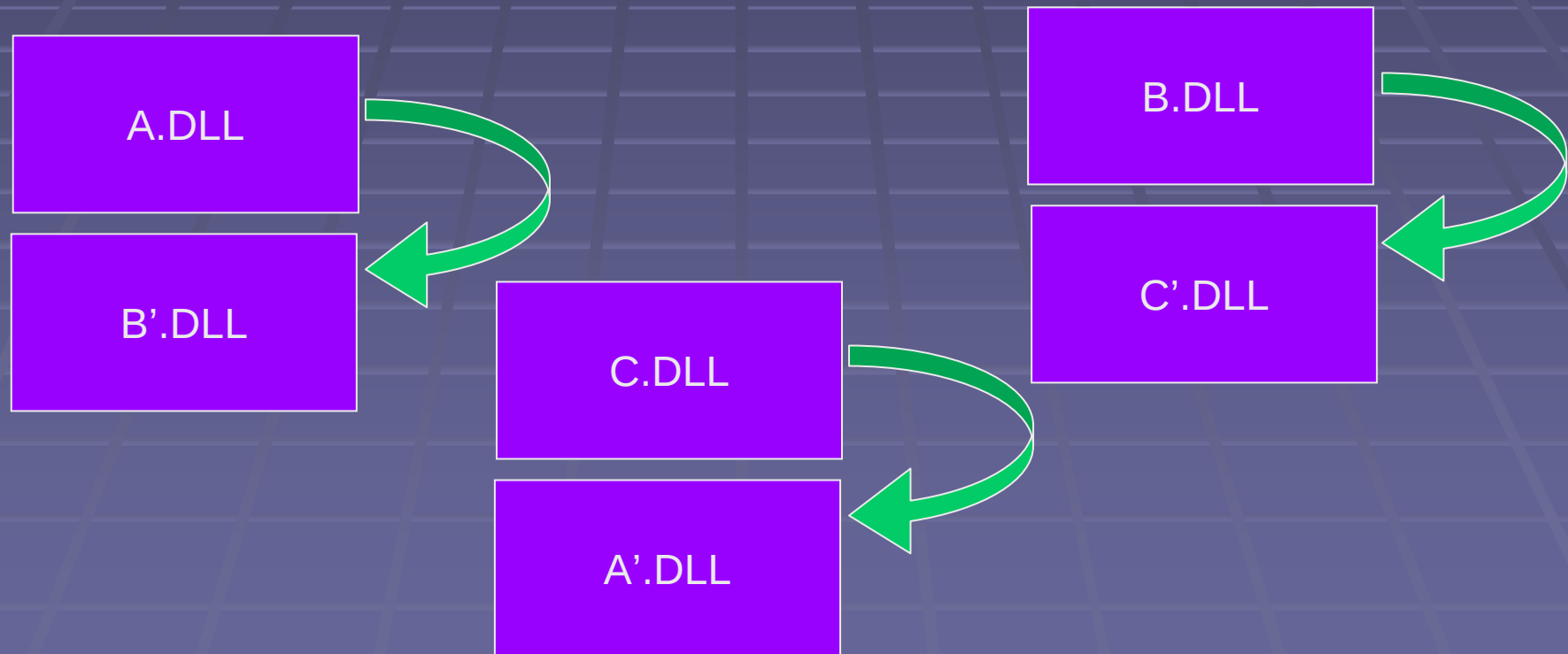
IL Assembler does not require the referenced assemblies to be present at compile time.

“Dummy” assemblies look exactly like “real” assemblies to C# or VB compiler



ASMMETA: Resolving Circular Dependencies

Step 3: No more circular dependencies. Build “real” assemblies

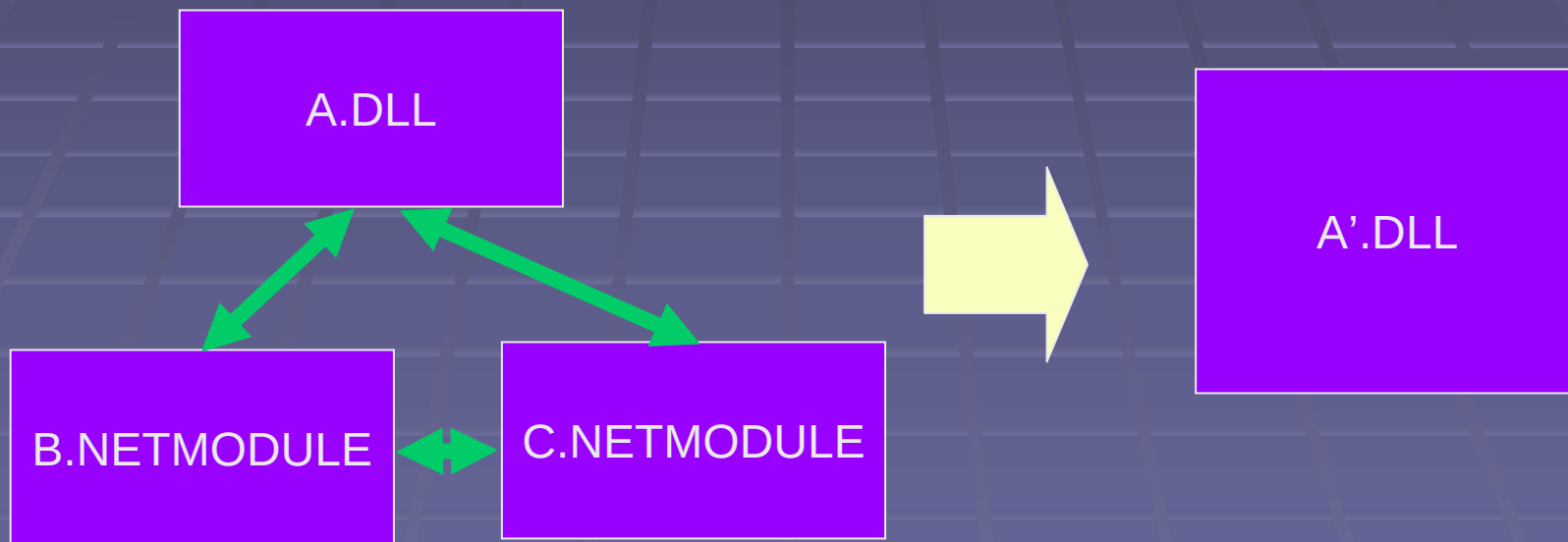


IL Assembler Technology Overview

- Build Environments Employing IL Assembler
 - ASMMETA: Resolving Circular Dependencies
 - ILLINK: Linking managed modules
 - Mike Stall's IL inlining in C# and VB
 - Preemptive's DotFuscator

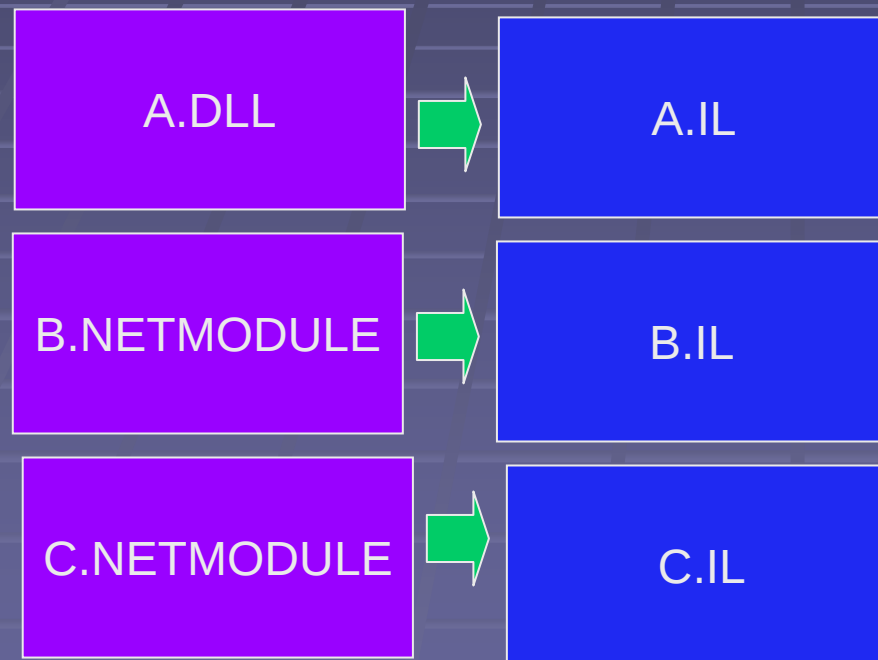
ILLINK: Linking managed modules

I have a multi-module assembly. How do I make a single-module assembly out of it?



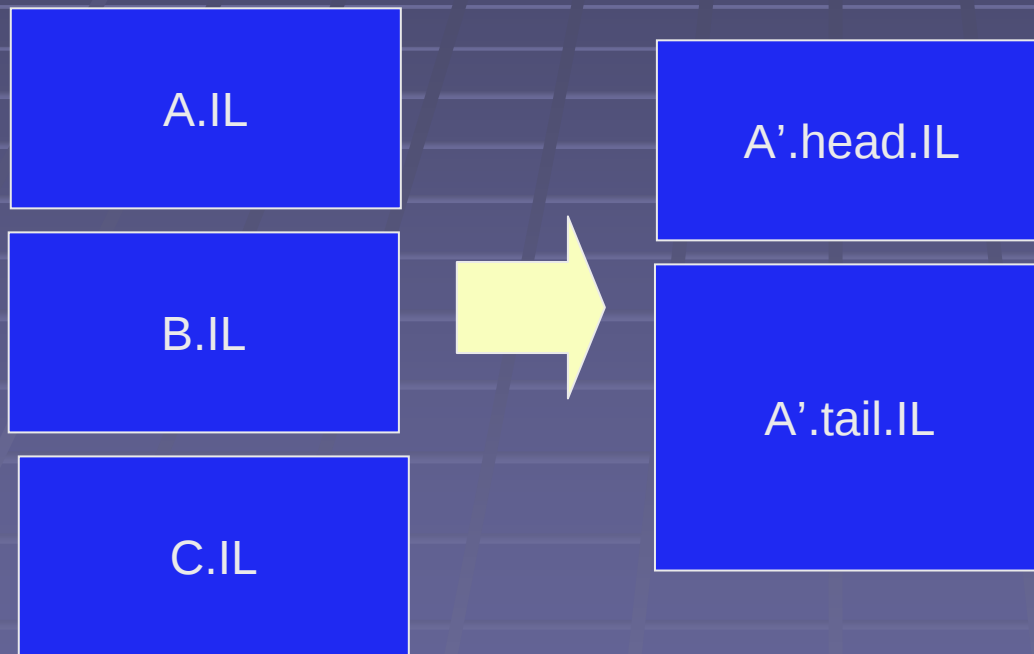
ILLINK: Linking managed modules

Step 1: Disassemble the modules



ILLINK: Linking managed modules

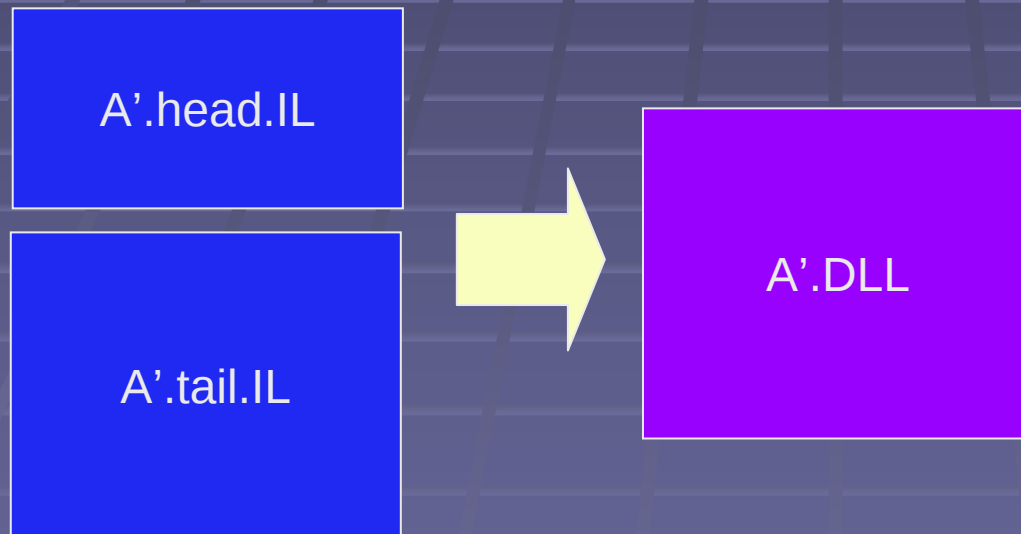
Step 2: Process the text



Text processing includes line reshuffling (all external references and manifest declarations go to “head”, all the rest goes to “tail”) and elimination of internal cross-references

ILLINK: Linking managed modules

Step 3: Assemble A'.DLL



ILLINK: Linking managed modules

ILLINK: 523 lines in C, including comments

<http://www.gotdotnet.com/community/usersamples/Default.aspx?query=ILLINK>

IL Assembler Technology Overview

- Build Environments Employing IL Assembler
 - ASMMETA: Resolving Circular Dependencies
 - ILLINK: Linking managed modules
 - Mike Stall's IL inlining in C# and VB
 - Preemptive's DotFuscator

Mike Stall's IL inlining in C# and VB

C# sample

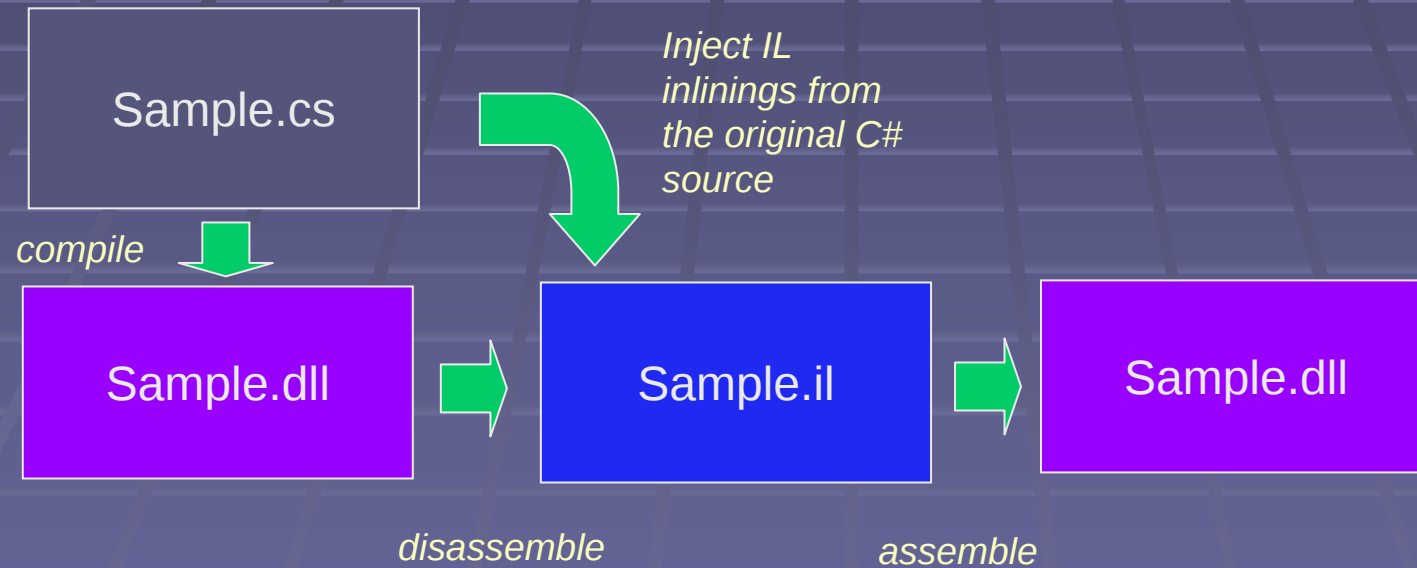
```
using System;
class Program
{
    static void Main()
    {
        int x = 3;
        int y = 4;
        int z = 5;
        // Here's some inline IL; "x=x+y+z"
#if IL
        ldloc x
        ldloc y
        add
        ldloc z
        add
        stloc x
#endif
        Console.WriteLine(x);
    }
}
```

VB sample

```
' VB demo with inline IL
module m1
sub Main()
    Console.WriteLine("Hi!")
    Dim x As Integer
    x = 3
    #If IL Then
        // Here's some inline IL
        ldloc x
        dup
        add
        stloc x
    #End If
    Console.WriteLine(x)
End Sub
end module
```

Mike Stall's IL inlining in C# and VB

How does it work?



Mike Stall's IL inlining in C# and VB

InlineIL: about 600 lines in C#, including
comments

<http://blogs.msdn.com/jmstall/archive/2005/02/21/377806.aspx>

IL Assembler Technology Overview

- Build Environments Employing IL Assembler
 - ASMMETA: Resolving Circular Dependencies
 - ILLINK: Linking managed modules
 - Mike Stall's IL inlining in C# and VB
 - Preemptive's DotFuscator

Preemptive's DotFuscat

What is obfuscation?

```
.class TelemetricData
{
    .field private int16 ChannelID
    .field private float32 AirSpeed
    .field private int64 Azimuth
    .field private int32 Altitude
    ...
    .method private bool TimeToTarget(
        int32 targetX,int32 targetY)
    {
        ...
    }
}
```



```
.class A
{
    .field private int16 A
    .field private float32 A
    .field private int64 A
    .field private int32 A
    ...
    .method private bool A(
        int32 targetX,int32 targetY)
    {
        ...
    }
}
```

Preemptive's DotFuscator

DotFuscator features:

- Types and members renaming;
- Control flow obfuscation;
- Unused types and members removal;
- String encryption;
- Software watermarking

<http://www.preemptive.com/products/dotfuscator/Features.html>

Preemptive's DotFuscator

How is it done?

- Disassemble the original assembly;
- Analyze the disassembly text;
- Alter the disassembly text;
- Re-assemble.

Sounds familiar?

Recent Developments

- Generics
- Custom Attribute encoding
- Declarative Security encoding
- Compilation control directives
- Reference aliasing (**.typedef** directive)
- Targeting 64 bit platforms
- Type forwarding
- IL Disassembler new features

Recent Developments

- Generics
- Custom Attribute encoding
- Declarative Security encoding
- Compilation control directives
- Reference aliasing (**.typedef** directive)
- Targeting 64 bit platforms
- Type forwarding
- IL Disassembler new features

Generics

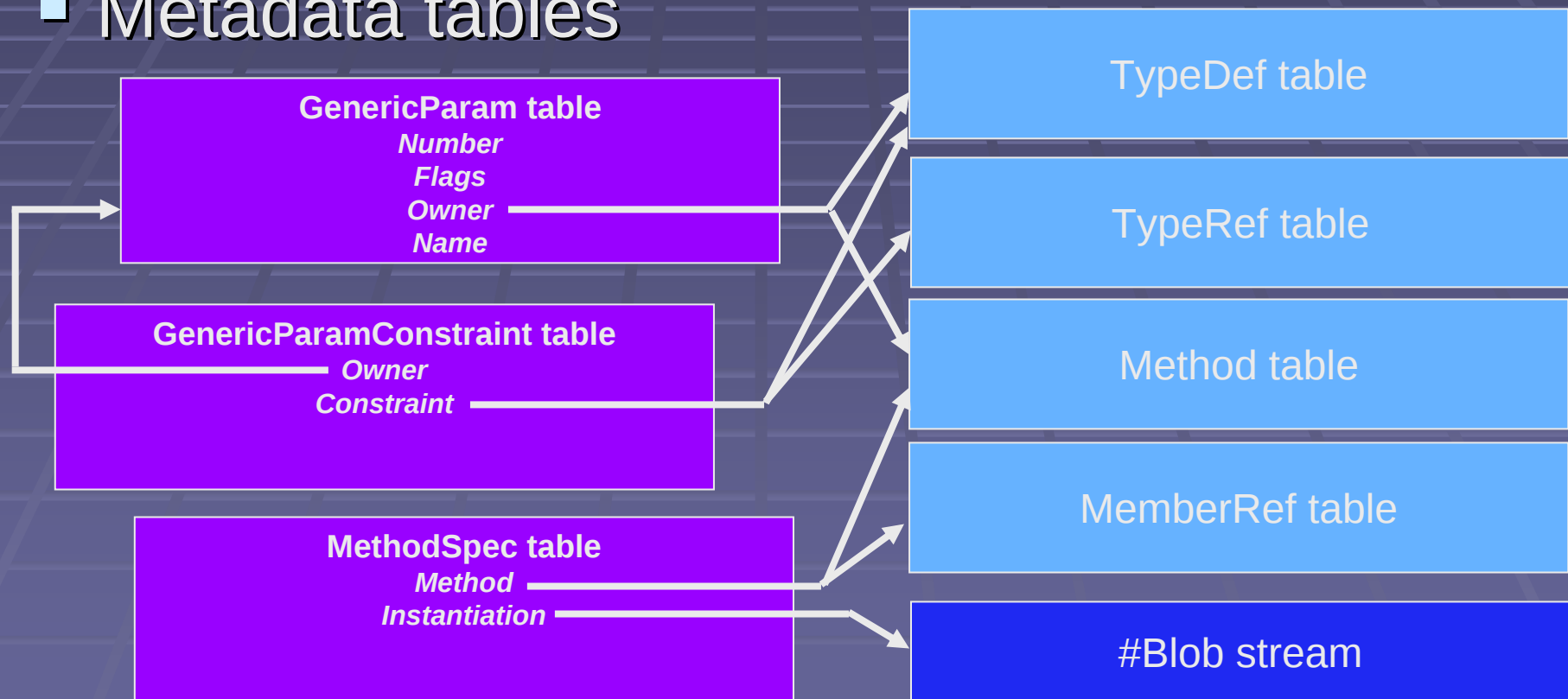
- What is it?
 - Type or method that has type parameter(s)
For example, stack of some type, sorter of some type, etc.
`stack<T>; void sort<U>(U array[], int arraySize, ...);`
 - You can constrain the type parameter(s), requiring that the prospective type argument had certain feature.
For example, `void sort<U:Comparable>(U array[],...);`
 - Instantiation of a generic type (method) is construction of a concrete type from a generic type by specifying type arguments
For example, `stack<string>;`
`void sort<double>(double array[],...);`
 - Functionality similar to C++ templates
 - C++ templates are resolved at compile time, generics are persisted in metadata and resolved at JIT compile time. Advantage: you can define a generic in one assembly and use it in dozen others.

Generics

- Metadata tables
 - GenericParam table
 - Type parameter information
 - Constraints “must be a reference type”, “must be a value type”, “must have a default constructor”, “covariant”, “contravariant”
 - GenericParamConstraint table
 - Constraints “must extend”, “must implement”
 - MethodSpec table
 - Generic method instantiations

Generics

- Metadata tables



Generics

- IL Assembler syntax
 - Declaring generic types

```
.class public Stack<T>
{
    .field private uint32 size
    .field private uint32 top
    .field private !T[] array
    .method public void Push(!T t)
    {
        ...
    }
    .method public !T Pop()
    {
        ...
    }
    ...
}
```

Generics

- IL Assembler syntax
 - Declaring generic methods

```
.class private System.Collections.Generic.ArraySortHelper<T>
{
    ...
    .method private void QuickSort<TValue>(!T[] keys,
                                           !!TValue[] values,
                                           int32 left,
                                           int32 right,
                                           class System.Collections.Generic.IComparer<!T> comparer)
    {
        ...
    }
    ...
}
```

Generics

- IL Assembler syntax
 - Constraining type parameters

```
// T must be a reference type
.class public Stack<class T>
{
    ...
}

// T must be a reference type and have a default constructor
.class public Copier<class .ctor T>
{
    ...
}

// T must be a reference type and implement IComparable
.class public SortedArray<class ([mscorlib]System.IComparable)T>
{
    ...
}
```

Generics

- IL Assembler syntax
 - Referencing generic type instantiations

```
.class public Stack<class T>
{
    ...
}
.class public FloatStackUser
{
    .field private class Stack<float32> stack
    .method public specialname void .ctor()
    {
        ...
        ldarg.0
        ldc.i4.s 64
        newobj instance void class Stack<float32>::.ctor(int32)
        stfld class Stack<float32> FloatStackUser::stack
        ...
    }
}
```

Generics

- IL Assembler syntax
 - Referencing generic method instantiations

```
.class public Stack<class T> { ... }  
...  
.class public A {  
    .method public static class Stack<!!T> StackFromArray<T>(!!T[] array) { ... }  
}  
.class public B {  
    .method public float32[] GetFloatArray() { ... }  
}  
...  
call instance float32[] B::GetFloatArray()  
call class Stack<!!0> A::StackFromArray<float32>(!!0[])  
...
```

Generics

- IL Assembler syntax
 - Nested generic types

```
.class public A<T>
{
    .class nested private Association<T,U>
    {
        ...
    }
    ...
}
```

Generics

- IL Assembler syntax
 - Nested generic types

```
// C#
class Tree<T> {
    class Node<T> { ... }
    ...
}

// IL
.class public Tree<T>
{
    .class nested private Node<T,T>
    {
        ...
    }
    ...
}
```

Recent Developments

- Generics
- Custom Attribute encoding
- Declarative Security encoding
- Compilation control directives
- Reference aliasing (**.typedef** directive)
- Targeting 64 bit platforms
- Type forwarding
- IL Disassembler new features

Custom Attribute encoding

// Binary form

```
.custom instance void [System]System.ComponentModel.BrowsableAttribute::.ctor(bool,
                                                                    object)
    = ( 01 00 01 08 39 30 00 00 05 00 53 50 03 42 61 72 // ....90....SP.Bar
        59 57 68 6F 2E 4B 6E 6F 77 73 2E 57 68 61 74 2C // YWho.Knows.What,
        20 53 79 73 74 65 6D 2C 20 56 65 72 73 69 6F 6E // System, Version
        ...
```

// Verbal form

```
.custom instance void [System]System.ComponentModel.BrowsableAttribute::.ctor(bool,
                                                                    object)
    = {bool(true)
        object(int32(12345))
        field type 'Bar' = type([System]Who.Knows.What)
        field object 'fooBar' = object(type([System]Never.Heard.Of))
        field int32[] 'fooBaz' = int32[3](1 2 3)
        field string[] 'fooFoo' = string[2]('abc' 'def')
        property string 'Foo' = string('He\\he\\he')}
```

Recent Developments

- Generics
- Custom Attribute encoding
- Declarative Security encoding
- Compilation control directives
- Reference aliasing (**.typedef** directive)
- Targeting 64 bit platforms
- Type forwarding
- IL Disassembler new features

Declarative security encoding

- **Permission set blob encoding**
 - *'.' // dot character*
 - *<compressed_uint> // number of permissions in the set*
 - *{ <permission> } // set of permission encodings*
- **Permission blob encoding**
 - *<compressed_uint> // length of the class name (following)*
 - *<class_name> // fully qualified class name in Reflection notation*
 - *<compressed_uint> // size of initialization blob*
 - *<compressed_uint> // number of properties, can be 0*
 - *[{ <property> }] // set of properties, absent if num=0*

Declarative security encoding

- **Permission property encoding**
 - `SERIALIZATION_TYPE_PROPERTY` // 1 byte, 0x54
 - `<type_of_the_property>` // property signature
 - `<compressed_uint>` // length of the property name (follows)
 - `<property_name>`
 - `<encoded_value>`
- **IL Assembler syntax**

```
.permissionset reqmin  
    = {[mscorlib]System.Security.Permissions.SecurityPermissionAttribute  
        = {property bool 'SkipVerification' = bool(true)}}
```

Recent Developments

- Generics
- Custom Attribute encoding
- Declarative Security encoding
- Compilation control directives
- Reference aliasing (.typedef directive)
- Targeting 64 bit platforms
- Type forwarding
- IL Disassembler new features

Compilation Control Directives

- `#include "<file name>"`
 - Include path can be specified by `/INCLUDE` command line option
- `#define <var_name>`
- `#define <var_name> "<string>"`
- `#ifdef <var_name>`
- `#ifndef <var_name>`
- `#else`
- `#endif`

Compilation Control Directives

```
// File AssemblyVer.il
#define VER_SYSTEM "2:0:0:0"
#define VER_MSCORLIB "2:0:0:0"
#define VER_VISUALB "8:0:0:0"
#define MS_PK_TOKEN_DCL ".publickeytoken = (B7 7A 5C 56 19 34 E0 89)"

// File MyProg.il
#include "AssemblyVer.il"
.assembly extern mscorlib {
    .ver VER_MSCORLIB
    MS_PK_TOKEN_DCL
}
.assembly extern Microsoft.VisualBasic {
    .ver VER_VISUALB
    .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
}
// continued ...
```

Compilation Control Directives

```
// File MyProg.il (continued)
#define ClassA_LocalDeclaration

.assembly MyProg { }
#ifdef ClassA_LocalDeclaration
.class public abstract A {
    .method public virtual abstract void a() {}
}
#else
    #include "ClassA.il"
#endif

.class public B extends A {
    .method public virtual void a() {
        ...
    }
    ...
}
```

Recent Developments

- Generics
- Custom Attribute encoding
- Declarative Security encoding
- Compilation control directives
- Reference aliasing (.typedef directive)
- Targeting 64 bit platforms
- Type forwarding
- IL Disassembler new features

Reference aliasing (.typedef directive)

- **Types**
 - .typedef [mscorlib]System.Console as TTY
 - .typedef string as str
- **Methods**
 - .typedef method void TTY::WriteLine(str) as PrintString
- **Fields**
 - .typedef field string A::hello as hithere
- **Custom Attributes**
 - .typedef .custom instance void [System]System.ComponentModel.BrowsableAttribute::.ctor(bool) = {bool(true)} as BrowsableAttr

Recent Developments

- Generics
- Custom Attribute encoding
- Declarative Security encoding
- Compilation control directives
- Reference aliasing (**.typedef** directive)
- Targeting 64 bit platforms
- Type forwarding
- IL Disassembler new features

Targeting 64 bit platforms

- Target classification

Target	Machine (COFF hdr)	NT Optional header type	Lower 2 bits of CLR header flags
Platform-agnostic	I386	32-bit	0x1
x86	I386	32-bit	0x0, 0x2, 0x3
Intel Itanium	IA64	64-bit	*
AMD X64	AMD64	64-bit	*

Targeting 64 bit platforms

- File structure checks for different targets
 - Headers:
 - DOS header: DOS Signature, pointer to COFF header
 - COFF header: Characteristics != SYSTEM
 - NT Optional Header:
 - NT Signature
 - Magic (32/64 bit)
 - FileAlignment (*512)
 - SectionAlignment (*FileAlignment)
 - SizeOfImage (*SectionAlignment)
 - ImageBase (*0x10000)
 - SizeOfStackCommit <= SizeOfStackReserve
 - SizeOfHeapCommit <= SizeOfHeapReserve
 - directory entries (for IL-only images, only IMPORT, RESOURCE, SECURITY, BASERELOC, DEBUG, IAT and COMHEADER are allowed)

Targeting 64 bit platforms

- File structure checks for different targets (continued)
 - Headers:
 - Section headers:
 - Characteristics (allowed: CODE, INITIALIZED DATA, UNINITIALIZED DATA, DISCARDABLE, NOT_CACHED, NOT_PAGED, EXECUTE, READ, WRITE, SHARED; ***not*** allowed: CODE|WRITE for all images and SHARED for IL-only images)
 - CLR header:
 - Read-only section
 - Metadata, resources and SN signature also in read-only section
 - IL-only images must not have VT Fixup table, EAT Jump table or Native Entrypoint;
 - Metadata headers:
 - Storage signature
 - on Intel Itanium, Metadata general header and storage header must be 4-byte aligned
 - storage header and streams must be contained within metadata size declared in CLR header
 - storage streams must not overlap

Targeting 64 bit platforms

- File structure checks for different targets (continued)
 - Relocations (specific to the platform: I386/Itanium/X64)
 - Import Address Table (in IL-only images, single entry pointing to mscoree.dll is allowed)

Targeting 64 bit platforms

- Different targets in IL Assembler
 - Target options
 - /PE64 – generate a PE32+ image
 - /ITAnuim – generate image for Intel Itanium
 - /X64 – generate image for AMD X64
 - Defaults:
 - /PE64 → /PE64 /ITA
 - /ITA → /PE64 /ITA
 - /X64 → /PE64 /X64
 - Automatic generation of VT Fixups for unmanaged exports

Targeting 64 bit platforms

```
.assembly extern mscorlib { }
.assembly YDD { }
.module YDD.dll
.corflags 0x00000002
.data VT_01 = int32(0)[3]

.vtfixup [3] int32 fromunmanaged at VT_01
.method public static int32 Yabba()
{
    .vtentry 1:1
    .export [1]
    ldstr "Printing Yabba (IL)"
    call void [mscorlib]System.Console::WriteLine(string)
    ldc.i4 0xAAAA
    ret
}
.method public static int64 Dabba()
{
    .vtentry 1:2
    .export [2]
    ldstr "Printing Dabba (IL)"
    call void [mscorlib]System.Console::WriteLine(string)
    ldc.i8 1234567890
    ret
}
.method public static float64 Doo()
{
    .vtentry 1:3
    .export [3]
    ldstr "Printing Doo! (IL)"
    call void [mscorlib]System.Console::WriteLine(string)
    ldc.r8 12.345678
    ret
}
```

Targeting 64 bit platforms

```
.assembly extern mscorlib { }
.assembly YDD { }
.module YDD.dll
// .corflags 0x00000002
// .data VT_01 = int32(0)[3]

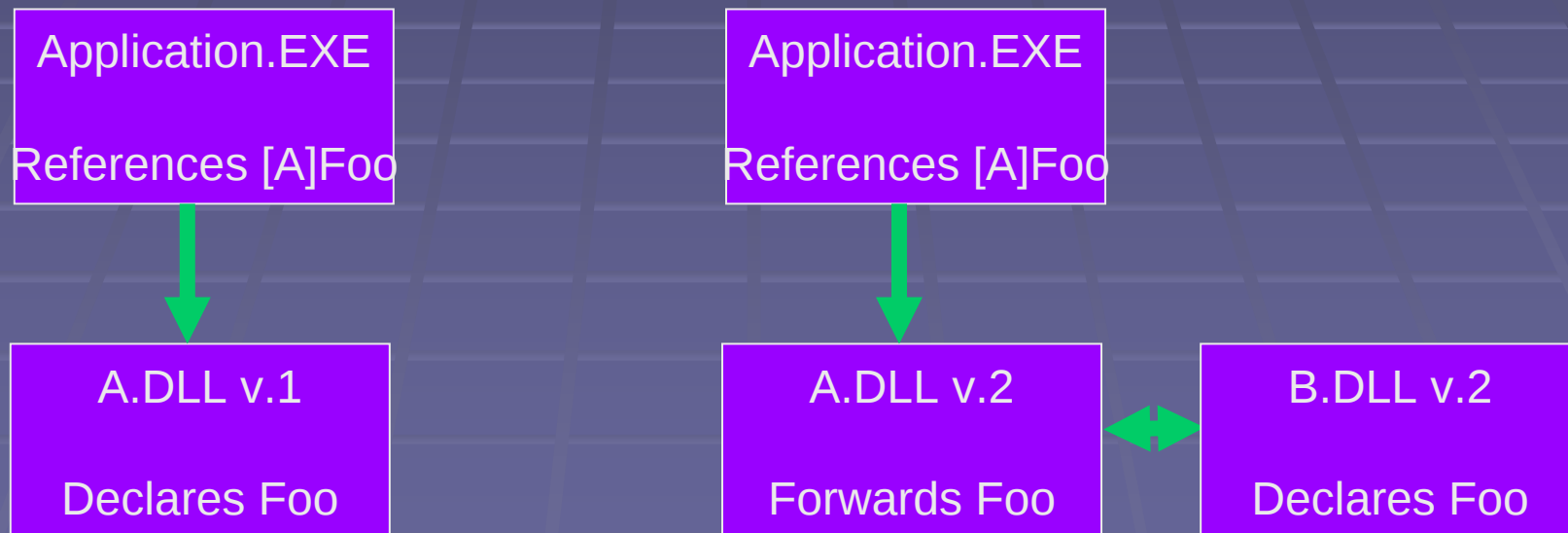
// .vtfixup [3] int32 fromunmanaged at VT_01
.method public static int32 Yabba()
{
    // .ventry 1:1
    .export [1]
    ldstr "Printing Yabba (IL)"
    call void [mscorlib]System.Console::WriteLine(string)
    ldc.i4 0xAAAA
    ret
}
.method public static int64 Dabba()
{
    // .ventry 1:2
    .export [2]
    ldstr "Printing Dabba (IL)"
    call void [mscorlib]System.Console::WriteLine(string)
    ldc.i8 1234567890
    ret
}
.method public static float64 Doo()
{
    // .ventry 1:3
    .export [3]
    ldstr "Printing Doo! (IL)"
    call void [mscorlib]System.Console::WriteLine(string)
    ldc.r8 12.345678
    ret
}
```

Recent Developments

- Generics
- Custom Attribute encoding
- Declarative Security encoding
- Compilation control directives
- Reference aliasing (**.typedef** directive)
- Targeting 64 bit platforms
- Type forwarding
- IL Disassembler new features

Type forwarding

- Assembly refactoring



Type forwarding

- Metadata presentation of type forwarders

- ExportedType table

- Flags | tdForwarder (0x00200000)
 - TypeDefId
 - TypeName
 - TypeNamespace
 - Implementation = mdAssemblyRef

- IL Assembler syntax

```
.class extern public forwarder Foo.Bar  
{  
    .assembly extern B  
}
```

Recent Developments

- Generics
- Custom Attribute encoding
- Declarative Security encoding
- Compilation control directives
- Reference aliasing (**.typedef** directive)
- Targeting 64 bit platforms
- Type forwarding
- IL Disassembler new features

IL Disassembler new features

- Files sample.il et al.
- No forward type declaration by default
- Typelist: preserving order of declared types
.typelist { name1 name2 ... nameN }
- References summary

```
//===== R E F E R E N C E S =====  
//      '<Module>'/*02000001*/  references  
//          - C/*02000004*/  
//          - A/*02000002*/  
//          - [mscorlib/*23000001*/]System.Console/*01000002*/  
//          - B/*02000003*/  
//      A/*02000002*/  references  
//          - [mscorlib/*23000001*/]System.Object/*01000001*/  
//          - C/*02000004*/  
//      B/*02000003*/  references  
//          - [mscorlib/*23000001*/]System.Object/*01000001*/  
//          - C/*02000004*/  
//      C/*02000004*/  references  
//          - [mscorlib/*23000001*/]System.Object/*01000001*/  
//===== END REFERENCES =====
```

Future Work

- **Componentization**
 - Make IL Assembler and IL Disassembler component-based rather than stand-alone tools
- **Nautilus Project**
 - A group within Visual Studio experimenting with development tools and environments written entirely in managed code. We are considering incorporating IL Assembler.
- **Phoenix Project**
 - A group within Visual Studio experimenting with universal code generation tools (John Lefor's presentation). We are considering incorporating IL Assembler and Disassembler and expanding them on Phoenix base.

- Questions?
- Contact: slidin@microsoft.com