Dynamic Types

CIL

Dynamic assemblies

Dynamic Types and DLR (Dynamic Language Runtime)

My comment:
Should only be used when absolutely necessary!!

- **dynamic keyword**

NOTICE:

- dynamic types are not dynamic assemblies!

dynamic types are **NOTstrongly typed**

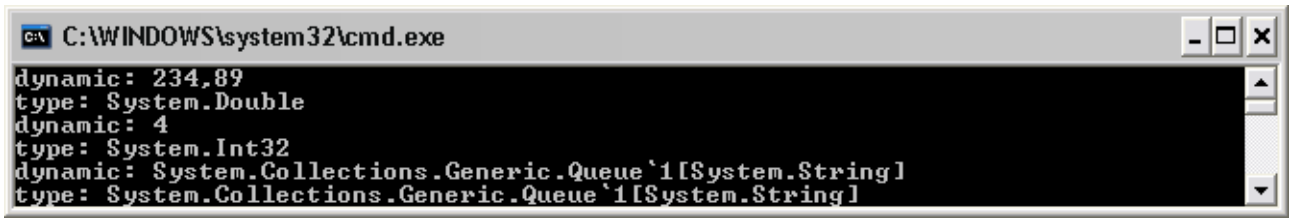
Intro example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DynamicTypesExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            dynamic no = 234.89;

            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());
            no = 4;
            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());



        }
    }
}
```

- Also works on generics

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DynamicTypesExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            dynamic no = 234.89;

            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());
            no = 4;
```

```
            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());
            no = new Queue<string>();

            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());




        }
    }
}
```

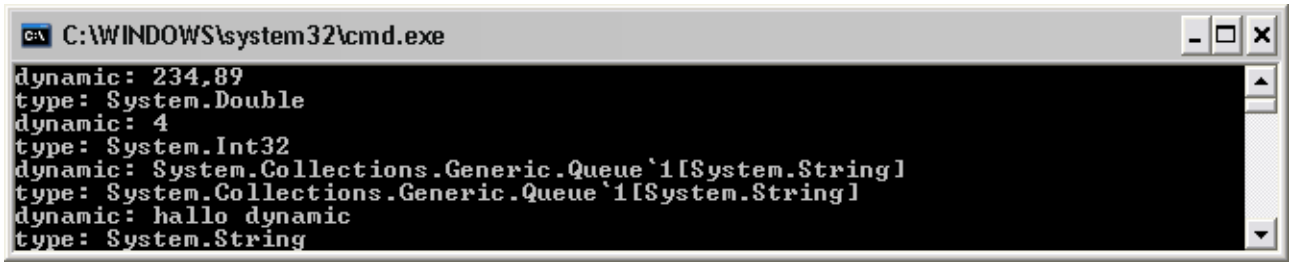


Methods on types

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DynamicTypesExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            dynamic no = 234.89;

            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());
            no = 4;
            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());


            no = new Queue<string>();

            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());

            no = "HALLO DYNAMIC";
            // Can I use ToLower() (defined in the string class) ??
            Console.WriteLine("dynamic: {0}", no.ToLower());
```

```
            Console.WriteLine("type: {0}", no.GetType());




        }
    }
}
```

- **Yes – that is possible**



Difference between var and dynamic:

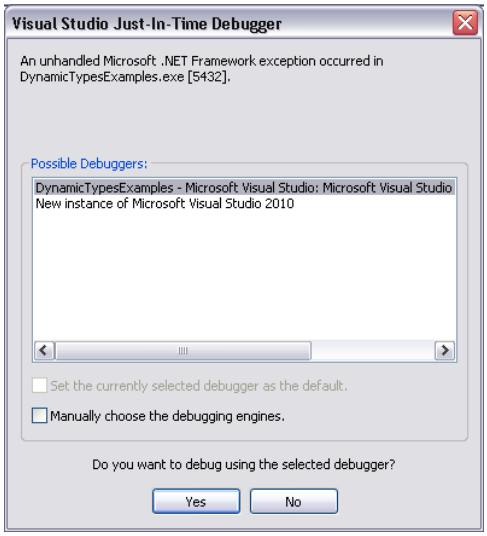
```
var mitObj "A string";
..

mitObj = 218;   // compiler error

..

dynamic mitObj "A string";
..

mitObj = 218;   // OK

..
```

It recreates the type of mitObj as an integer when we assign an integer value to mitObj.

Type conversion?

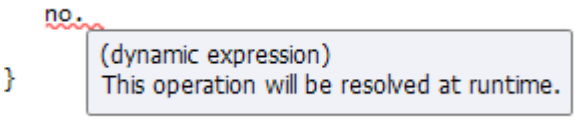
```
no = (int)no;
```

- **DEPENDS ON THE TYPE LEFT HAND VARIABLE**



No – Not working!
(Error on runtime – NOT compile time)

- Visual Studio has no intellisense for dynamic types (of course)
- Watch out: Incorrect member name gives crash on runtime



Exception handling is a must

- Use the RuntimeBinder class
- in the Microsoft.CSharp.dll assembly

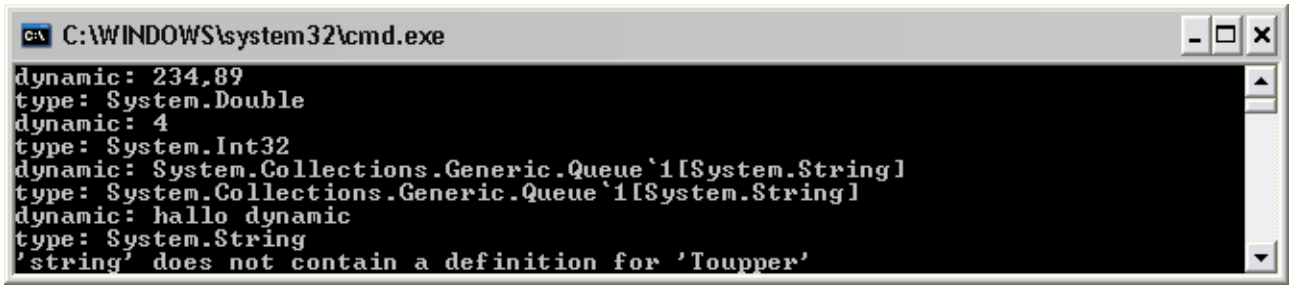
Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace DynamicTypesExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            dynamic no = 234.89;

            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());
            no = 4;
            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());


            no = new Queue<string>();

            Console.WriteLine("dynamic: {0}", no);
            Console.WriteLine("type: {0}", no.GetType());

            no = "HALLO DYNAMIC";

            Console.WriteLine("dynamic: {0}", no.ToLower());
            Console.WriteLine("type: {0}", no.GetType());


           try
           {
             no = (int)no;

           Console.WriteLine("dynamic: {0}", no);
           Console.WriteLine("type: {0}", no.GetType());

           }catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException mcrre)
           {
               Console.WriteLine(mcrre.Message);
           }




        }
    }
}
```

- Will also catch wrong method call

```
...
     try
           {
           Console.WriteLine("dynamic: {0}", no.Toupper()); // Toupper() ???
           Console.WriteLine("dynamic: {0}", no);
           Console.WriteLine("type: {0}", no.GetType());

           }catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException mcrre)
           {
               Console.WriteLine(mcrre.Message);
           }
...
```



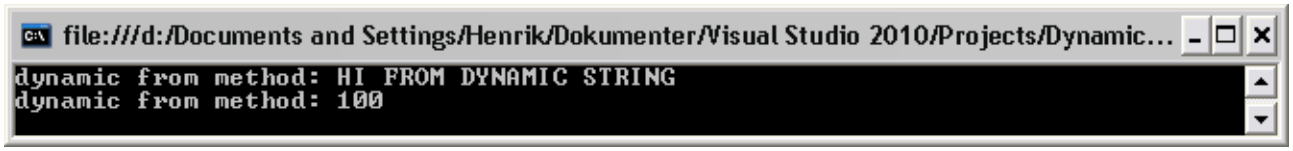
dynamic keyword scope

- a dynamic can be a return value from a method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DynamicTypesExamples
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            dynamic a = "hi from ";
            dynamic b = "dynamic string";
            dynamic c = ADynamicMethod(a, b);
            Console.WriteLine("dynamic from method: {0}", c);

            a = 12;
            b = 88;
            c = ADynamicMethod(a, b);
            Console.WriteLine("dynamic from method: {0}", c);


        }

        public static dynamic ADynamicMethod(dynamic a, dynamic b)
        {
            if (a is string & b is string )
              return (a + b).ToUpper();
            if (a is int & b is int)
                return a + b;
            else
                return "NO TYPE FOUND";


        }
    }
}
```

Output



Limitations

- dynamics cannot use
  - lambda expression
  - anonymous methods

Where can dynamic types be used?

- Well suited with late binding
- .NET applications that communicates with COM libraries

DLR (Dynamic Language Runtime)

- Known principle in Python, SmallTalk and other languages
- C# uses NORMAL strongly typed types
- Focus is on flexibility
- Addition and removal of types on run-time

DLR uses an expansion tree

```
dynamic d = GetSomeData();
d.SuperMethod(12);
```

- Build an expansion tree
- Calls the SuperMethod with 12 as an argument
- Passed on to the correct Runtime Binder finding a method suitable for this

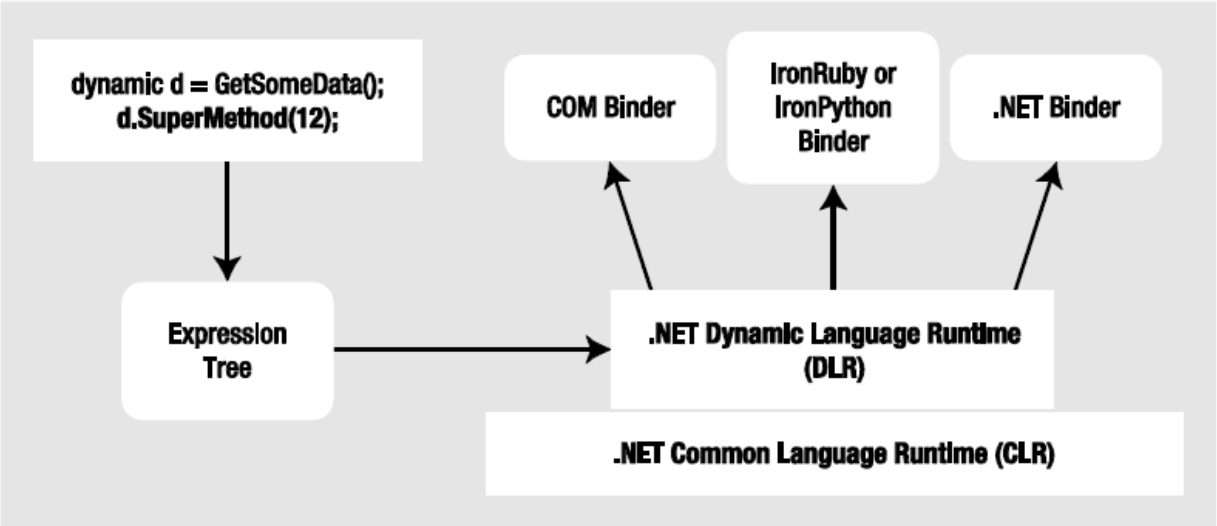


Figure 16-3. *Expression trees capture dynamic calls in neutral terms and are processed by binders*

Another small example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace DynamicTypesExamples
{
    class Program
    {
        static void Main(string[] args)
        {

            dynamic a = "hi from ";
            dynamic b = "dynamic string";
            dynamic c = ADynamicMethod(a, b);
            Console.WriteLine("dynamic from method: {0}", c);

            a = 12;
            b = 88;
            c = ADynamicMethod(a, b);
            Console.WriteLine("dynamic from method: {0}", c);
            dynamic d;
            int c_dt = 0;
            int c_s = 0;
            for (int i = 0; i < 100; i++)
            {
                d = Dyn1();
                if (d is DateTime)
                    c_dt++;
                else if (d is string)
                    c_s++;

            }

            Console.WriteLine("No of DateTime {0}", c_dt);
            Console.WriteLine("No of Strings {0}", c_s);







        }

        public static dynamic ADynamicMethod(dynamic a, dynamic b)
        {
            if (a is string & b is string )
              return (a + b).ToUpper();
            if (a is int & b is int)
```

```
                return a + b;
            else
                return "NO TYPE FOUND";


        }

        public static dynamic Dyn1()
        {
            Thread.Sleep(123);
            DateTime dt = DateTime.Now;
            if (dt.Millisecond % 2 == 0)
                return dt;
            else
                return "0.1";
        }

    }
}
```

Output:

- Use of dynamics
  - dynamic used to create an object
  - Notice: Invoke is not used – object created directly

Dynamics can be used with late binding

While this is code works as expected, you might agree it is a bit clunky. Here, you have to manually make use of the `MethodInfo` class, manually query the metadata, and so forth. The following is a version of this same method, now using the C# `dynamic` keyword and the DLR:

```
static void InvokeMethodWithDynamicKeyword(Assembly asm)
{
  try
  {
    // Get metadata for the Minivan type.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");

    // Create the Minivan on the fly and call method!
    dynamic obj = Activator.CreateInstance(miniVan);
    obj.TurboBoost();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

dynamic data and COM

- COM is access to advanced components
  - example:
    - Word
    - Excel
    - AND MUCH MORE

- Remember to ”Add Reference” to the specific COM library

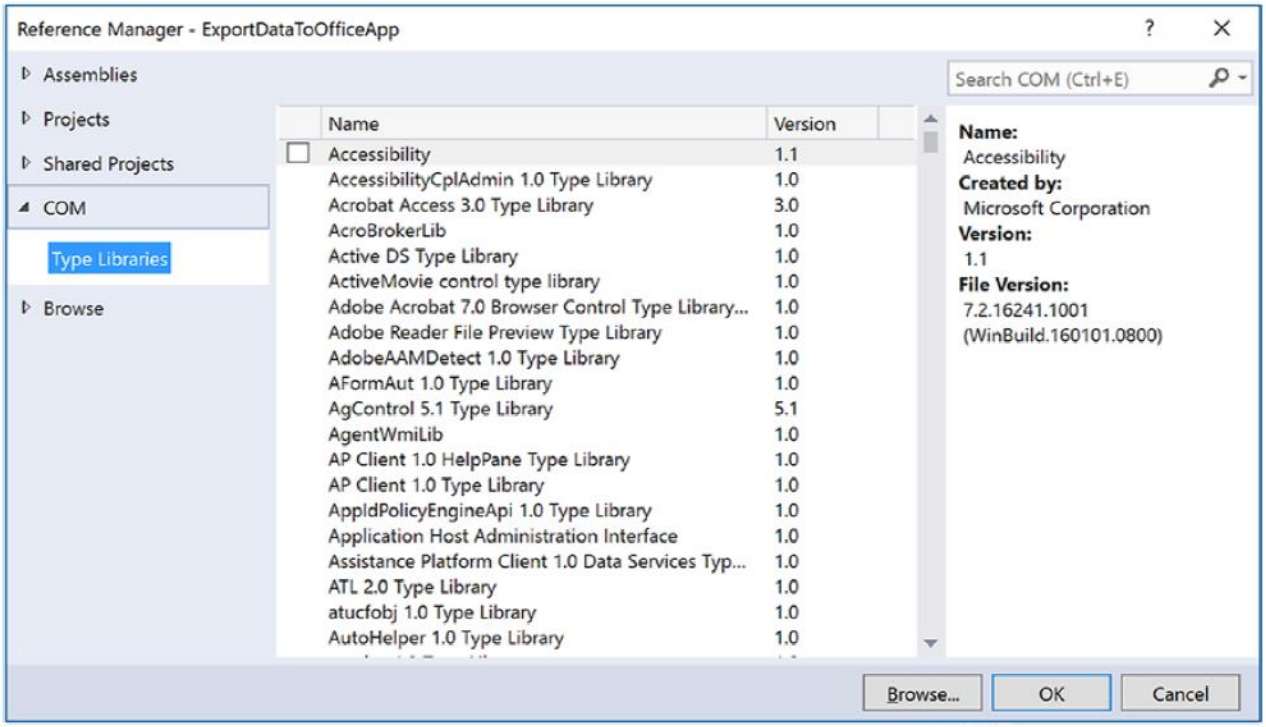


Figure 16-4. *The COM tab of the Add Reference dialog box will show you all registered COM libraries on your machine*

Communication with COM objects using a proxy

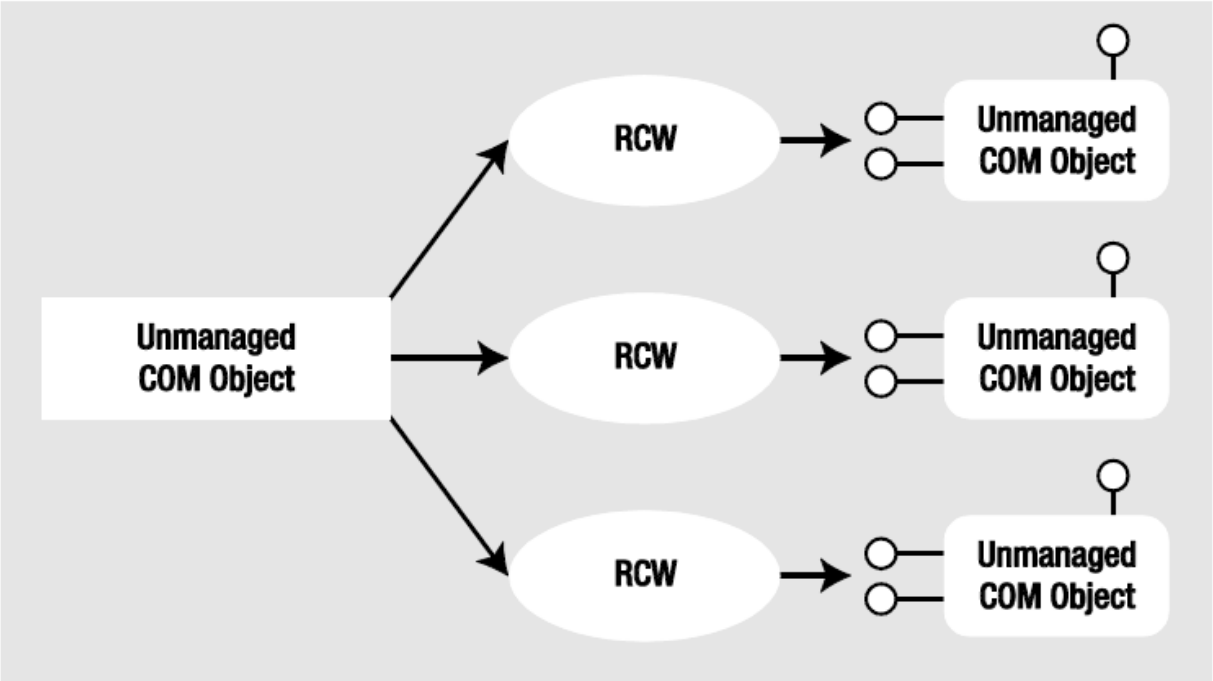
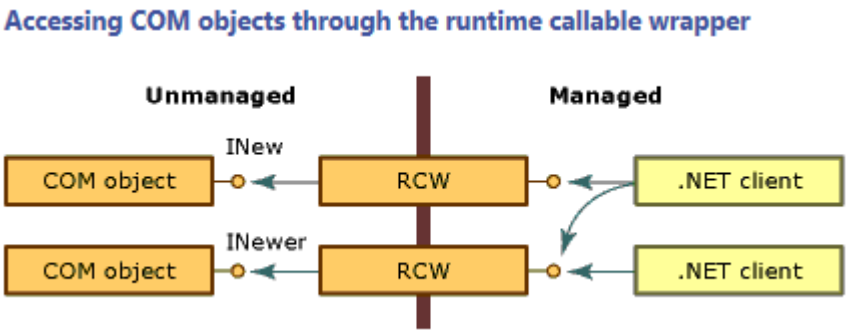


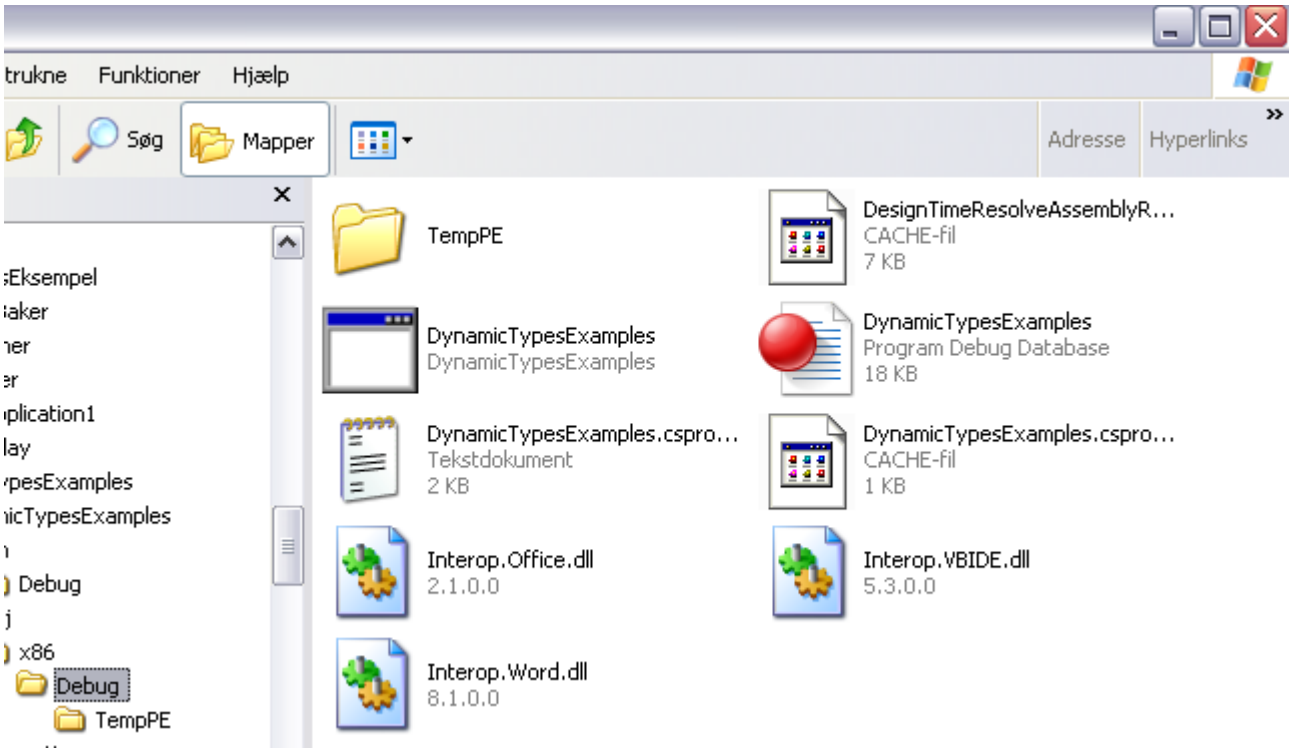
Figure 16-6. *.NET programs communicate with COM objects using a proxy termed the RCW*

RCW

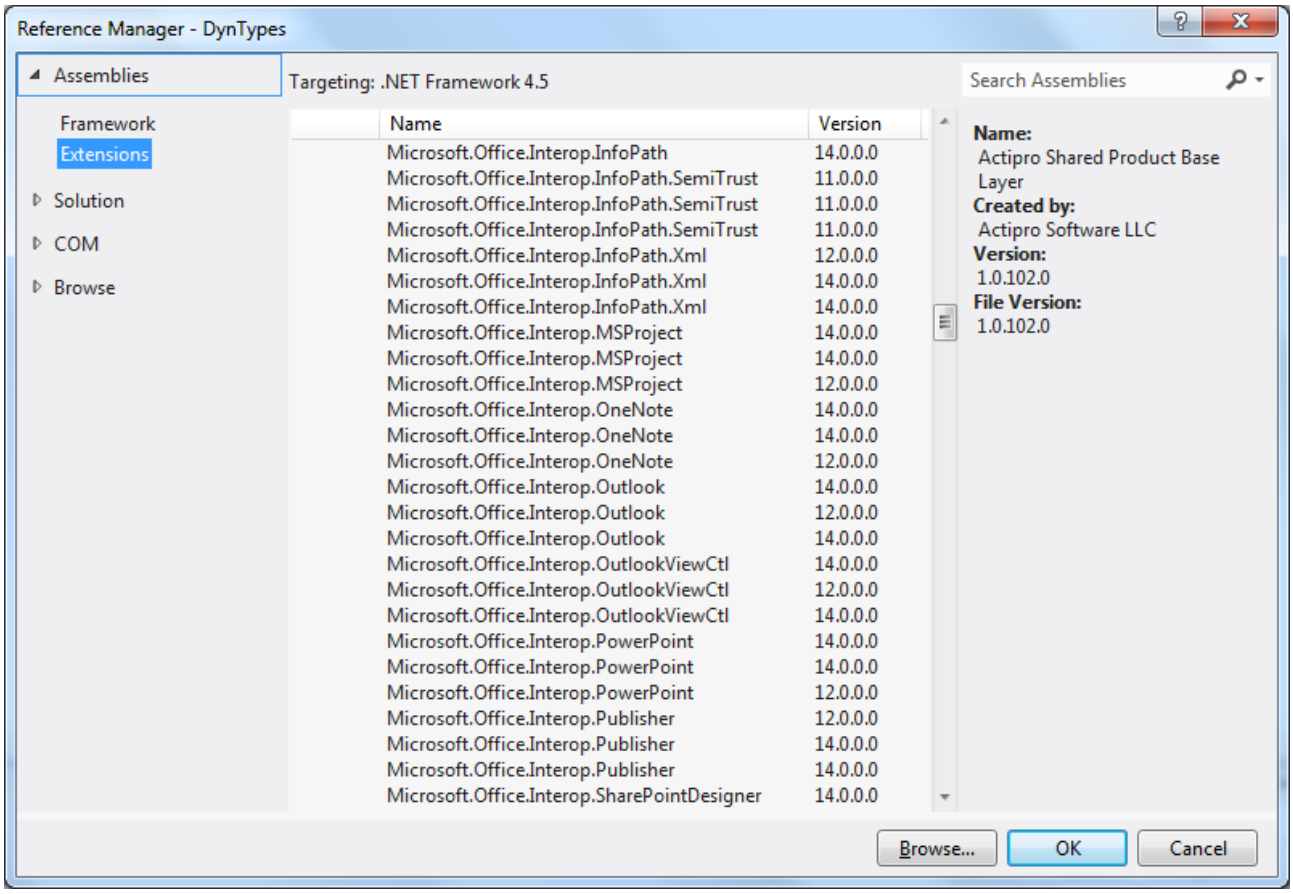


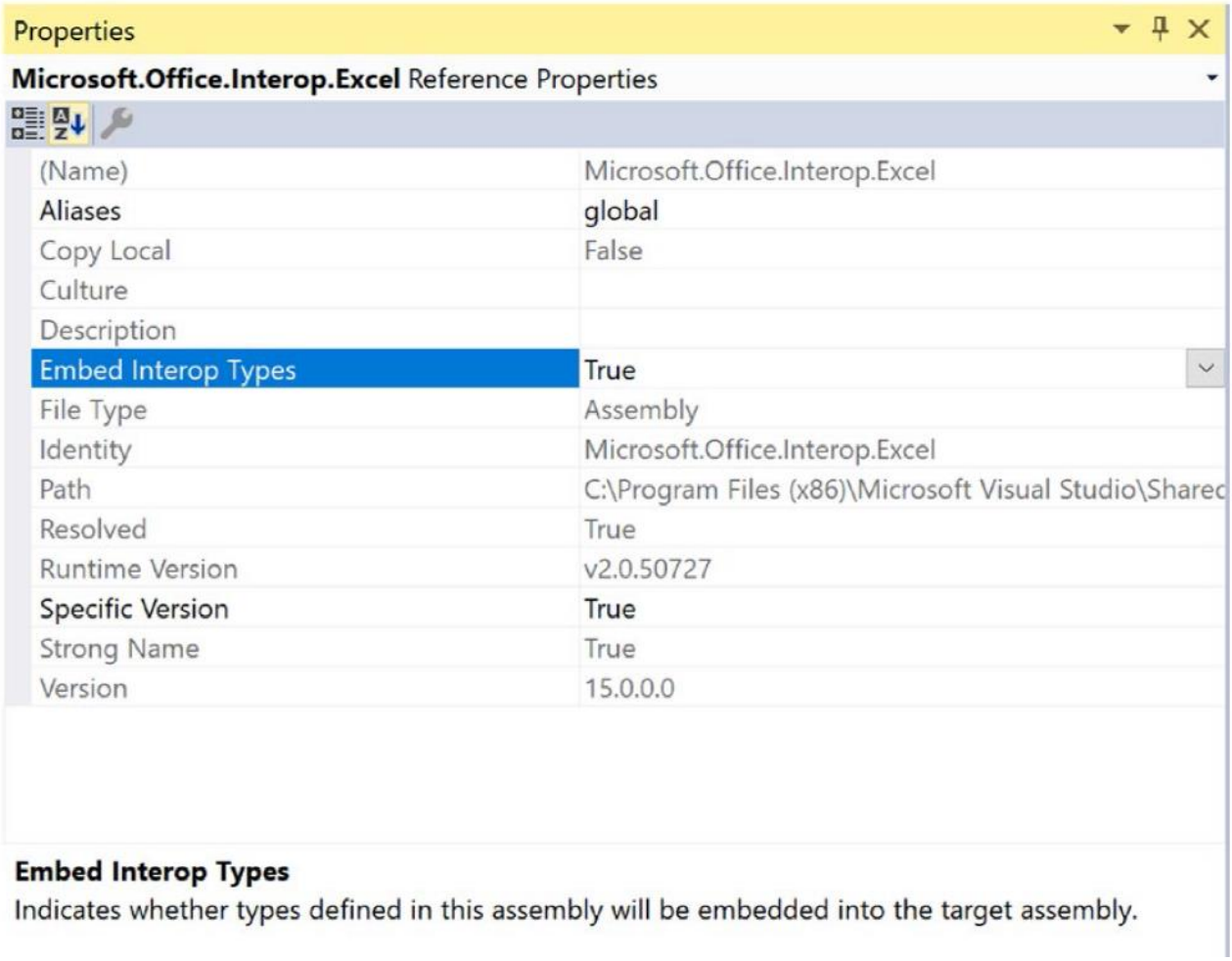
Primary Interop Assemblies (PIA)

- Many producers allows access to their specialized assemblies using
  - Primary Interop Assemblies (relatively easy access)
- Earlier: The assemblies were placed in GAC
- Since 4.0 direct access with a copy



- Extension tab:


Reference Manager - DynTypes
Assemblies
Framework
Extensions
Solution
COM
Browse
Targeting: .NET Framework 4.5
Search Assemblies
Name
Version
Microsoft.Office.Interop.InfoPath 14.0.0.0
Microsoft.Office.Interop.InfoPath.SemiTrust 11.0.0.0
Microsoft.Office.Interop.InfoPath.SemiTrust 11.0.0.0
Microsoft.Office.Interop.InfoPath.SemiTrust 11.0.0.0
Microsoft.Office.Interop.InfoPath.Xml 12.0.0.0
Microsoft.Office.Interop.InfoPath.Xml 14.0.0.0
Microsoft.Office.Interop.InfoPath.Xml 14.0.0.0
Microsoft.Office.Interop.MSProject 14.0.0.0
Microsoft.Office.Interop.MSProject 14.0.0.0
Microsoft.Office.Interop.MSProject 12.0.0.0
Microsoft.Office.Interop.OneNote 14.0.0.0
Microsoft.Office.Interop.OneNote 14.0.0.0
Microsoft.Office.Interop.OneNote 12.0.0.0
Microsoft.Office.Interop.Outlook 14.0.0.0
Microsoft.Office.Interop.Outlook 12.0.0.0
Microsoft.Office.Interop.Outlook 14.0.0.0
Microsoft.Office.Interop.OutlookViewCtl 14.0.0.0
Microsoft.Office.Interop.OutlookViewCtl 12.0.0.0
Microsoft.Office.Interop.OutlookViewCtl 14.0.0.0
Microsoft.Office.Interop.PowerPoint 14.0.0.0
Microsoft.Office.Interop.PowerPoint 14.0.0.0
Microsoft.Office.Interop.PowerPoint 12.0.0.0
Microsoft.Office.Interop.Publisher 12.0.0.0
Microsoft.Office.Interop.Publisher 14.0.0.0
Microsoft.Office.Interop.Publisher 14.0.0.0
Microsoft.Office.Interop.SharePointDesigner 14.0.0.0
Name:
Actipro Shared Product Base Layer
Created by:
Actipro Software LLC
Version:
1.0.102.0
File Version:
1.0.102.0
Browse...
OK
Cancel

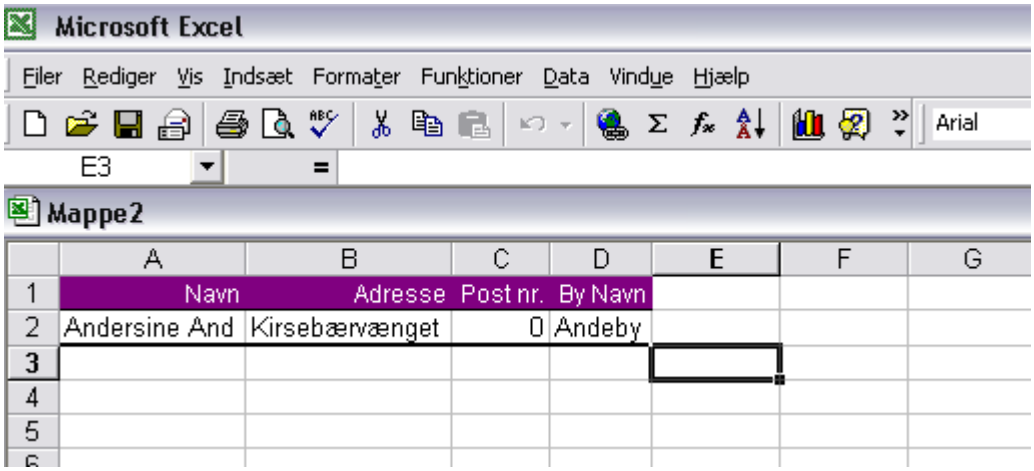


Interop and Type.Missing

- Earlier the ”Type.Missing” were used many-many places in the code
- Now the code should be simpler, see page 660
- Uses a kind of ”Skip”

Example: Create an Excel sheet:

```
public static void CreateExcel()
   {
       Excel.Application excelApp = new Excel.Application();
       excelApp.Workbooks.Add();

       Excel._Worksheet workSheet = excelApp.ActiveSheet;

       workSheet.Cells[1, "A"] = "Navn";
       workSheet.Cells[1, "B"] = "Adresse";
       workSheet.Cells[1, "C"] = "Post nr.";
       workSheet.Cells[1, "D"] = "By Navn";

       workSheet.Cells[2, "A"] = "Andersine And";
       workSheet.Cells[2, "B"] = "Kirsebærvænget";
       workSheet.Cells[2, "C"] = "0000";
       workSheet.Cells[2, "D"] = "Andeby";

       string environ = Environment.CurrentDirectory;
       workSheet.SaveAs(string.Format(@"{0}\person.xlsx",environ));

       excelApp.Quit();


   }
```



Example 2: Word (Not the newest Word version!)

(Example from msdn)

```
public static void WordDocCreator()
      {

          object oMissing = System.Reflection.Missing.Value;
          object oEndOfDoc = "\\endofdoc"; /* \endofdoc is a predefined bookmark */

          //Start Word and create a new document.
          Word._Application oWord;
          Word._Document oDoc;
          oWord = new Word.Application();
          oWord.Visible = true;
          oDoc = oWord.Documents.Add(ref oMissing, ref oMissing,
              ref oMissing, ref oMissing);

          //Insert a paragraph at the beginning of the document.
          Word.Paragraph oPara1;
          oPara1 = oDoc.Content.Paragraphs.Add(ref oMissing);
          oPara1.Range.Text = "Heading 1";
          oPara1.Range.Font.Bold = 1;
          oPara1.Format.SpaceAfter = 24;    //24 pt spacing after paragraph.
          oPara1.Range.InsertParagraphAfter();

          //Insert a paragraph at the end of the document.
          Word.Paragraph oPara2;
          object oRng = oDoc.Bookmarks.Item(ref oEndOfDoc).Range;
          oPara2 = oDoc.Content.Paragraphs.Add(ref oRng);
          oPara2.Range.Text = "Heading 2";
          oPara2.Format.SpaceAfter = 6;
          oPara2.Range.InsertParagraphAfter();

          //Insert another paragraph.
          Word.Paragraph oPara3;
          oRng = oDoc.Bookmarks.Item(ref oEndOfDoc).Range;
          oPara3 = oDoc.Content.Paragraphs.Add(ref oRng);
          oPara3.Range.Text = "This is a sentence of normal text. Now here is a table:";
          oPara3.Range.Font.Bold = 0;
          oPara3.Format.SpaceAfter = 24;
          oPara3.Range.InsertParagraphAfter();

          //Insert a 3 x 5 table, fill it with data, and make the first row
          //bold and italic.

          Word.Table oTable;
          Word.Range wrdRng = oDoc.Bookmarks.Item(ref oEndOfDoc).Range;
          oTable = oDoc.Tables.Add(wrdRng, 3, 5, ref oMissing, ref oMissing);
          oTable.Range.ParagraphFormat.SpaceAfter = 6;
          int r, c;
          string strText;
          for (r = 1; r <= 3; r++)
              for (c = 1; c <= 5; c++)
              {
                  strText = "r" + r + "c" + c;
                  oTable.Cell(r, c).Range.Text = strText;
              }
```

```
        Word.Paragraph oPara4;
        oRng = oDoc.Bookmarks.Item(ref oEndOfDoc).Range;
        oPara4 = oDoc.Content.Paragraphs.Add(ref oRng);
        oPara4.Range.InsertParagraphBefore();
        oPara4.Range.Text = "And here's another table:";
        oPara4.Format.SpaceAfter = 24;
        oPara4.Range.InsertParagraphAfter();

        //Insert a 5 x 2 table, fill it with data, and change the column widths.
        wrdRng = oDoc.Bookmarks.Item(ref oEndOfDoc).Range;
        oTable = oDoc.Tables.Add(wrdRng, 5, 2, ref oMissing, ref oMissing);
        oTable.Range.ParagraphFormat.SpaceAfter = 6;
        for (r = 1; r <= 5; r++)
            for (c = 1; c <= 2; c++)
            {
                strText = "r" + r + "c" + c;
                oTable.Cell(r, c).Range.Text = strText;
            }


        //Keep inserting text. When you get to 7 inches from top of the
        //document, insert a hard page break.
        Object oPos;
        double dPos = oWord.InchesToPoints(7);
        oDoc.Bookmarks.Item(ref oEndOfDoc).Range.InsertParagraphAfter();
        do
        {
            wrdRng = oDoc.Bookmarks.Item(ref oEndOfDoc).Range;
            wrdRng.ParagraphFormat.SpaceAfter = 6;
            wrdRng.InsertAfter("A line of text");
            wrdRng.InsertParagraphAfter();
            oPos = wrdRng.get_Information
                        (Word.WdInformation.wdVerticalPositionRelativeToPage);
        }
        while (dPos >= Convert.ToDouble(oPos));
        object oCollapseEnd = Word.WdCollapseDirection.wdCollapseEnd;
        object oPageBreak = Word.WdBreakType.wdPageBreak;
        wrdRng.Collapse(ref oCollapseEnd);
        wrdRng.InsertBreak(ref oPageBreak);
        wrdRng.Collapse(ref oCollapseEnd);
        wrdRng.InsertAfter("We're now on page 2. Here's my chart:");
        wrdRng.InsertParagraphAfter();

        //Insert a chart.
        Word.InlineShape oShape;
        object oClassType = "MSGraph.Chart.8";
        wrdRng = oDoc.Bookmarks.Item(ref oEndOfDoc).Range;
        oShape = wrdRng.InlineShapes.AddOLEObject(ref oClassType, ref oMissing,
            ref oMissing, ref oMissing, ref oMissing,
            ref oMissing, ref oMissing, ref oMissing);

                //Add text after the chart.
        wrdRng = oDoc.Bookmarks.Item(ref oEndOfDoc).Range;
        wrdRng.InsertParagraphAfter();
        wrdRng.InsertAfter("THE END.");


    }
```

Output:

1. One Word document (Just below!)

Heading 1

Heading 2

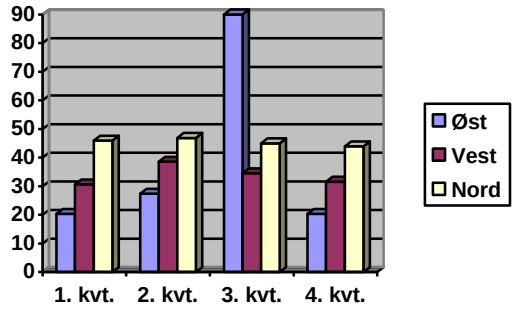
This is a sentence of normal text. Now here is a table:

r1c1	r1c2	r1c3	r1c4	r1c5
r2c1	r2c2	r2c3	r2c4	r2c5
r3c1	r3c2	r3c3	r3c4	r3c5

And here's another table:

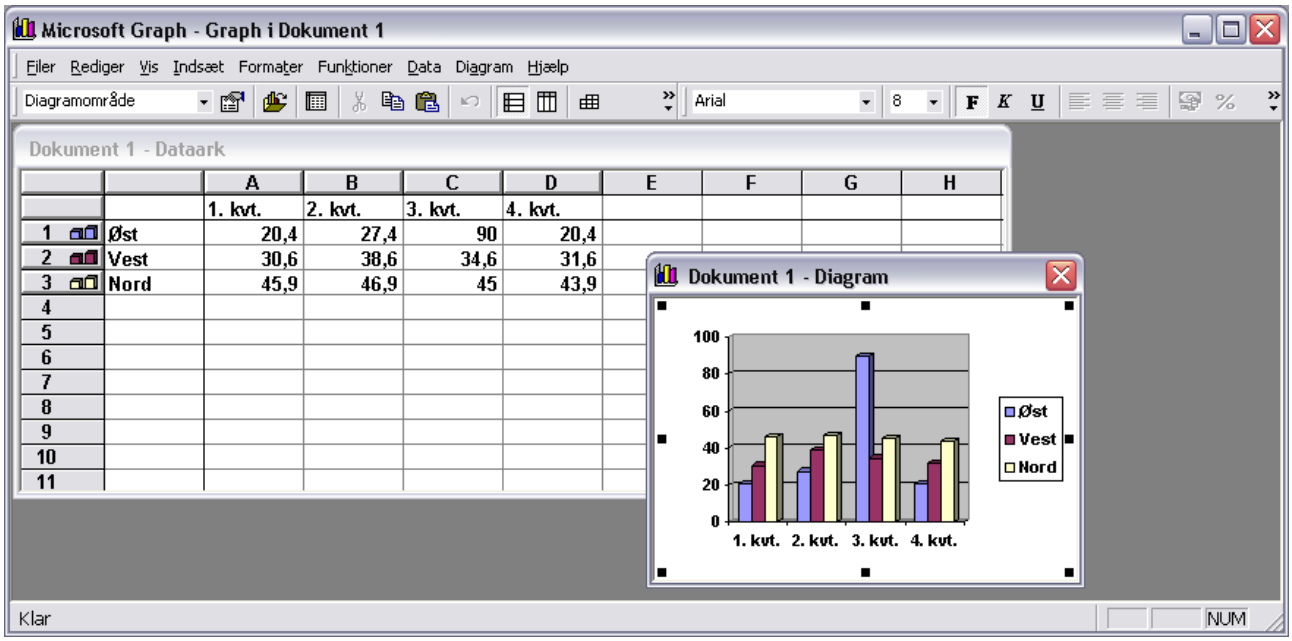
r1c1	r1c2
r2c1	r2c2
r3c1	r3c2
r4c1	r4c2
r5c1	r5c2

A line of text
A line of text
A line of text
A line of text
We're now on page 2. Here's my chart:



THE END.

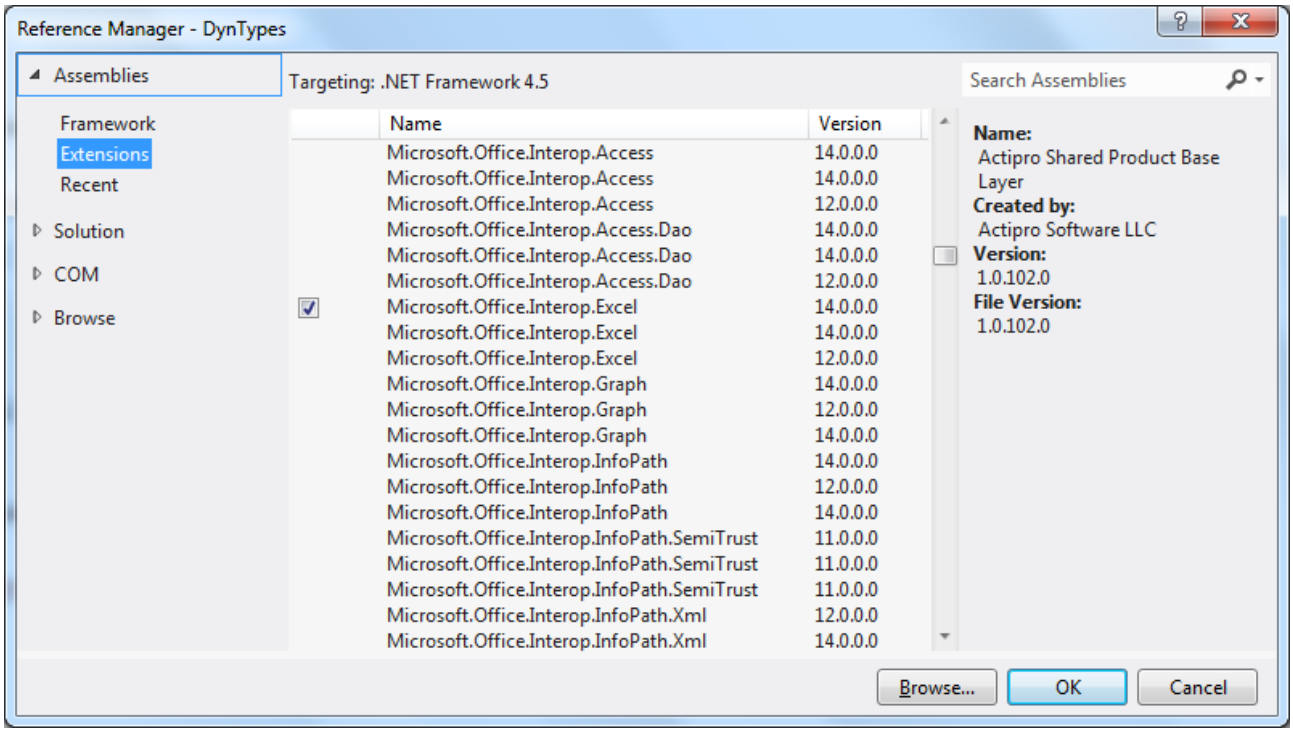
2. Microsoft Graph



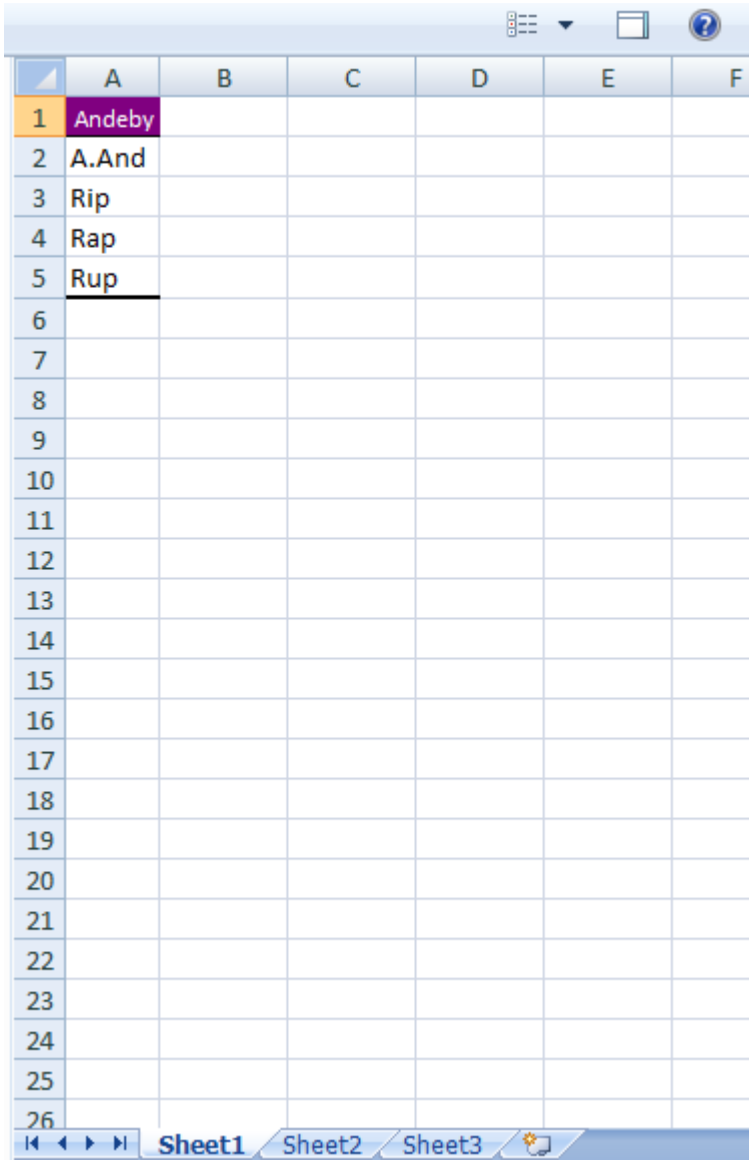
- Example: xlsx Excel

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Excel = Microsoft.Office.Interop.Excel;

namespace DynTypes
{
    class Program
    {

        private static List<String> info = new List<string>();

        static void Main(string[] args)
        {
            info.Add("A.And");
            info.Add("Rip");
            info.Add("Rap");
            info.Add("Rup");

            Excel.Application excelApp = new Excel.Application();
            excelApp.Workbooks.Add();

            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            workSheet.Cells[1, "A"] = "Andeby";

            int row = 1;
            foreach (String s in info)
            {

                workSheet.Cells[++row, "A"] = s;

            }
            //refine the sheet

workSheet.Range["A1"].AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
            // save it

            workSheet.SaveAs(string.Format(@"{0}\Andeby.xlsx",
                                                    Environment.CurrentDirectory));

        }
    }
}
```

- Reference to:


Reference Manager - DynTypes
Assemblies
Framework
Extensions
Recent
Solution
COM
Browse
Targeting: .NET Framework 4.5
Search Assemblies
Name
Version
Microsoft.Office.Interop.Access 14.0.0.0
Microsoft.Office.Interop.Access 14.0.0.0
Microsoft.Office.Interop.Access 12.0.0.0
Microsoft.Office.Interop.Access.Dao 14.0.0.0
Microsoft.Office.Interop.Access.Dao 14.0.0.0
Microsoft.Office.Interop.Access.Dao 12.0.0.0
Microsoft.Office.Interop.Excel 14.0.0.0
Microsoft.Office.Interop.Excel 14.0.0.0
Microsoft.Office.Interop.Excel 12.0.0.0
Microsoft.Office.Interop.Graph 14.0.0.0
Microsoft.Office.Interop.Graph 12.0.0.0
Microsoft.Office.Interop.Graph 14.0.0.0
Microsoft.Office.Interop.InfoPath 14.0.0.0
Microsoft.Office.Interop.InfoPath 12.0.0.0
Microsoft.Office.Interop.InfoPath 14.0.0.0
Microsoft.Office.Interop.InfoPath.SemiTrust 11.0.0.0
Microsoft.Office.Interop.InfoPath.SemiTrust 11.0.0.0
Microsoft.Office.Interop.InfoPath.SemiTrust 11.0.0.0
Microsoft.Office.Interop.InfoPath.Xml 12.0.0.0
Microsoft.Office.Interop.InfoPath.Xml 14.0.0.0
Name:
Actipro Shared Product Base Layer
Created by:
Actipro Software LLC
Version:
1.0.102.0
File Version:
1.0.102.0
Browse...
OK
Cancel

- Output:


A
B
C
D
E
F
1 Andeby
2 A.And
3 Rip
4 Rap
5 Rup
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
Sheet1
Sheet2
Sheet3

CIL and dynamic assemblies

- There exists
  - CIL directive
  - CIL attribute
  - CIL Opcode

  - Many arguments to have knowledge about CIL
    - Read page 662

CIL Directives

- Represented with a . (dot)
- namespace
- class
- publickeytoken
- method
- assembly (and a lot more directives)

CIL Attributtes

- If a directive is not fully expressed
- Example: public attribute on a class
- extends attribute (inheritance)
- implements attribute (interface)

CIL Opcode

- Operation Code
- All keyword have corresponding opcodes
- Example: **ldstr** is textual representation of a binary opcode (a number)

Push – Pop

- CIL is stack based
- There is a **.maxstack directive** (used inside methods)
- ldstr (or other ld.. command) places a value on stack
- which for instance is popped by a method
- The Stack must be 0 (zero) when the method is leaved
- Otherwise is the **pop command** used
- method is leaved with **ret**

CIL is not executed directly: It is compiles by the JIT on command. JIT optimize the CIL code in a smart way

Compiling CIL Code

Prior versions of .NET allowed you to compile *.il files using ilasm.exe. This has changed in .NET Core. To compile *.il files, you have to use a Microsoft.NET.Sdk.IL project type, and at the time of this writing, this is not part of the standard SDK.

Start by creating a new directory on your machine. In this directory, create a global.json file. The global.json file applies to the current directory and all subdirectories below the file. It is used to define which SDK version you will use when running .NET Core Command Line Interface (CLI) commands. Update the files to the following:

```
{
  "msbuild-sdks": {
    "Microsoft.NET.Sdk.IL": "3.0.0-preview1-27101-02"
  }
}
```

Note The current version (at the time of this writing) is still a preview version. Update your version to match the version at this URL: https://dotnet.myget.org/feed/dotnet-core/package/nuget/Microsoft.NET.Sdk.IL.

Next, add a NuGet.config to add in the additional MyGet location for the IL SDK and to confirm the location of the .NET Core NuGet feed. Update the file to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://www.nuget.org/api/v3/index.json" />
    <add key="DotnetCore" value="https://dotnet.myget.org/F/dotnet-core/api/v3/index.json" />
  </packageSources>
</configuration>
```

The final setup step is to create the project file. Create a file named RoundTrip.ilproj and update it to the following:

```
<Project Sdk="Microsoft.NET.Sdk.IL">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
<MicrosoftNetCoreIlasmPackageVersion>3.0.0-preview1-27101-02</
MicrosoftNetCoreIlasmPackageVersion>
  </PropertyGroup>
</Project>
```

NEXT:

Finally, copy in your updated RoundTrip.il file into the directory. Compile the assembly using the .NET Core CLI:

```
dotnet build
```

You will find the resulting files in the usual bin\debug\netcoreapp3.1 folder. At this point, you can run your new application. Sure enough, you will see the updated message displaying in the console window. While the output of this simple example is not all that spectacular, it does illustrate one practical use of programming in CIL: round-tripping.

Example CIL:

- My own CIL code (Notepad technology):

```
.assembly extern mscorlib{}

.assembly Hejsa
{

 .ver 1:0:1:0
}

.module Hejsa.exe

.method static void Main() cil managed
{
  .maxstack 1
  .entrypoint

  ldstr "Hejsa fra IL"
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

Hejsa.il

- Comments in CIL code as in C# and C++

Version

```
...
.assembly extern System.Runtime
{

 .ver 1:0:1:0
}

...
```

Major version 1
Minor version 0
Build 1
Revision 0

- . (DOT) indicates a directive

- Keywords do not use DOT notation

Classes

- Notice that classes must be used on in for instance C# !
- Can do without on CIL level

Additional directives

- Additionals

Table 18-2. *Additional Assembly-Centric Directives*

Directive	Meaning in Life
.mresources	If your assembly makes use of internal resources (such as bitmaps or string tables), this directive is used to identify the name of the file that contains the resources to be embedded.
.subsystem	This CIL directive is used to establish the preferred UI that the assembly wishes to execute within. For example, a value of 2 signifies that the assembly should run within a GUI application, whereas a value of 3 denotes a console executable.

Namespaces in CIL

Defining Namespaces in CIL

Now that you have defined the look and feel of your assembly (and the required external references), you can create a .NET Core namespace (`MyNamespace`) using the `.namespace` directive, like so:

```
// Our assembly has a single namespace.
.namespace MyNamespace {}
```

Like C#, CIL namespace definitions can be nested within further namespaces. There is no need to define a root namespace here; however, for the sake of argument, assume you want to create the following root namespace named `MyCompany`:

```
.namespace MyCompany
{
  .namespace MyNamespace {}
}
```

Like C#, CIL allows you to define a nested namespace as follows:

```
// Defining a nested namespace.
.namespace MyCompany.MyNamespace {}
```

Classes in CIL

Defining Class Types in CIL

Empty namespaces are not very interesting, so let's now check out the process of defining a class type using CIL. Not surprisingly, the `.class` directive is used to define a new class. However, this simple directive can be adorned with numerous additional attributes, to further qualify the nature of the type. To illustrate, add a public class to your namespace named `MyBaseClass`. As in C#, if you do not specify an explicit base class, your type will automatically be derived from `System.Object`.

```
.namespace MyNamespace
{
  // System.Object base class assumed.
  .class public MyBaseClass {}
}
```

Add full name

```
// This will not compile!
.namespace MyNamespace
{
  .class public MyBaseClass {}

  .class public MyDerivedClass
    extends MyBaseClass {}
}
```

To correctly define the parent class of `MyDerivedClass`, you must specify the full name of `MyBaseClass` as follows:

```
// Better!
.namespace MyNamespace
{
  .class public MyBaseClass {}

  .class public MyDerivedClass
    extends MyNamespace.MyBaseClass {}
}
```

Attributes in .class directives

Table 19-2. Various Attributes Used in Conjunction with the `.class` Directive

Attributes	Meaning in Life
`public`, `private`, `nested assembly`, `nested famandassem`, `nested family`, `nested famorassem`, `nested public`, `nested private`	CIL defines various attributes that are used to specify the visibility of a given type. As you can see, raw CIL offers numerous possibilities other than those offered by C#. Refer to ECMA 335 for details if you are interested.
`abstract`, `sealed`	These two attributes may be tacked onto a `.class` directive to define an abstract class or sealed class, respectively.
`auto`, `sequential`, `explicit`	These attributes are used to instruct the CLR how to lay out field data in memory. For class types, the default layout flag (`auto`) is appropriate. Changing this default can be helpful if you need to use P/Invoke to call into unmanaged C code.
`extends`, `implements`	These attributes allow you to define the base class of a type (via `extends`) or implement an interface on a type (via `implements`).

Defining and Implementing Interfaces in CIL

As odd as it might seem, interface types are defined in CIL using the `.class` directive. However, when the `.class` directive is adorned with the `interface` attribute, the type is realized as a CTS interface type. Once an interface has been defined, it may be bound to a class or structure type using the CIL `implements` attribute, like so:

```
.namespace MyNamespace
{
  // An interface definition.
  .class public interface IMyInterface {}

  // A simple base class.
  .class public MyBaseClass {}

  // MyDerivedClass now implements IMyInterface,
  // and extends MyBaseClass.
  .class public MyDerivedClass
    extends MyNamespace.MyBaseClass
    implements MyNamespace.IMyInterface {}
}
```

Note The `extends` clause must precede the `implements` clause. As well, the `implements` clause can incorporate a comma-separated list of interfaces.

- Notice the interface definition!

See definitions of

- Structures
- Enums
- Generics
- Page 675-677

Compiling the il types

Compiling the CILTypes.il File

Even though you have not yet added any members or implementation code to the types you have defined, you are able to compile this `*.il` file into a .NET Core DLL assembly (which you must do, as you have not specified a `Main()` method). Open a command prompt and enter the following command:

```
dotnet build
```

After you have done so, you can now open your compiled assembly into `ildasm.exe` to verify the creation of each type. To understand how to populate a type with content, you first need to examine the fundamental data types of CIL.

Mapping .NET Base Class Types to C# Keywords and C# Keywords to CIL

Table 19-3. *Mapping .NET Base Class Types to C# Keywords and C# Keywords to CIL*

.NET Core Base Class Type	C# Keyword	CIL Representation	CIL Constant Notation
System.SByte	Sbyte	int8	I1
System.Byte	Byte	unsigned int8	U1
System.Int16	Short	int16	I2
System.UInt16	Ushort	unsigned int16	U2
System.Int32	Int	int32	I4
System.UInt32	Uint	unsigned int32	U4
System.Int64	Long	int64	I8
System.UInt64	Ulong	unsigned int64	U8
System.Char	Char	char	CHAR
System.Single	Float	float32	R4
System.Double	Double	float64	R8
System.Boolean	Bool	bool	BOOLEAN
System.String	String	string	N/A
System.Object	Object	object	N/A
System.Void	Void	void	VOID

Note The `System.IntPtr` and `System.UIntPtr` types map to native `int` and native `unsigned int` (this is good to know, as many of COM interoperability and P/Invoke scenarios use these extensively).

Defining Field Data in CIL

Enumerations, structures, and classes can all support field data. In each case, the `.field` directive will be used. For example, let's breathe some life into the skeleton `MyEnum` enumeration and define the following three name-value pairs (note the values are specified within parentheses):

```
.class public sealed enum MyEnum
{
  .field public static literal valuetype
   MyNamespace.MyEnum A = int32(0)
  .field public static literal valuetype
   MyNamespace.MyEnum B = int32(1)
```

Another example

```
.class public MyBaseClass
{
  .field private string stringField = "hello!"
  .field private int32 intField = int32(42)
}
```

As in C#, class field data will automatically be initialized to an appropriate default value. If you want to allow the object user to supply custom values at the time of creation for each of these points of private field data, you (of course) need to create custom constructors.

Constructors: Notice the difference!

```
.class public MyBaseClass
{
  .field private string stringField
  .field private int32 intField

  .method public hidebysig specialname rtspecialname
    instance void .ctor(string s, int32 i) cil managed
  {
    // TODO: Add implementation code...
  }
}
```

Properties

Defining Properties in CIL

Properties and methods also have specific CIL representations. By way of an example, if `MyBaseClass` were updated to support a public property named `TheString`, you would author the following CIL (note again the use of the `specialname` attribute):

```
.class public MyBaseClass
{
...
  .method public hidebysig specialname
    instance string get_TheString() cil managed
  {
    // TODO: Add implementation code...
  }

  .method public hidebysig specialname
    instance void set_TheString(string 'value') cil managed
  {
    // TODO: Add implementation code...
  }

  .property instance string TheString()
  {
    .get instance string
      MyNamespace.MyBaseClass::get_TheString()
    .set instance void
      MyNamespace.MyBaseClass::set_TheString(string)
  }
}
```

In terms of CIL, a property maps to a pair of methods that take `get_` and `set_` prefixes. The `.property` directive makes use of the related `.get` and `.set` directives to map property syntax to the correct "specially named" methods.

Method parameters

To illustrate the process of defining parameters in raw CIL, assume you want to build a method that takes an `int32` (by value), an `int32` (by reference), a `[mscorlib]System.Collection.ArrayList`, and a single output parameter (of type `int32`). In terms of C#, this method would look something like the following:

```
public static void MyMethod(int inputInt,
  ref int refInt, ArrayList ar, out int outputInt)
{
  outputInt = 0; // Just to satisfy the C# compiler...
}
```

If you were to map this method into CIL terms, you would find that C# reference parameters are marked with an ampersand (&) suffixed to the parameter's underlying data type (`int32&`).

Output parameters also use the & suffix, but they are further qualified using the CIL `[out]` token. Also notice that if the parameter is a reference type (in this case, the `[mscorlib]System.Collections.ArrayList` type), the `class` token is prefixed to the data type (not to be confused with the `.class` directive!).

```
.method public hidebysig static void MyMethod(int32 inputInt,
  int32& refInt,
  class [System.Runtime.Extensions]System.Collections.ArrayList ar,
  [out] int32& outputInt) cil managed
{
```

Opcodes

Examining CIL Opcodes

The final aspect of CIL code you'll examine in this chapter has to do with the role of various operational codes (opcodes). Recall that an opcode is simply a CIL token used to build the implementation logic for a given member. The complete set of CIL opcodes (which is fairly large) can be grouped into the following broad categories:

- Opcodes that control program flow
- Opcodes that evaluate expressions
- Opcodes that access values in memory (via parameters, local variables, etc.)

To provide some insight to the world of member implementation via CIL, Table 19-4 defines some of the more useful opcodes that are directly related to member implementation logic, grouped by related functionality.

Table 19-4. *Various Implementation-Specific CIL Opcodes*

Opcodes	Meaning in Life
`add`, `sub`, `mul`, `div`, `rem`	These CIL opcodes allow you to add, subtract, multiply, and divide two values (`rem` returns the remainder of a division operation).
`and`, `or`, `not`, `xor`	These CIL opcodes allow you to perform bit-wise operations on two values.
`ceq`, `cgt`, `clt`	These CIL opcodes allow you to compare two values on the stack in various manners. Here are some examples: `ceq`: Compare for equality `cgt`: Compare for greater than `clt`: Compare for less than
`box`, `unbox`	These CIL opcodes are used to convert between reference types and value types.
`ret`	This CIL opcode is used to exit a method and return a value to the caller (if necessary).
`beq`, `bgt`, `ble`, `blt`, `switch`	These CIL opcodes (in addition to many other related opcodes) are used to control branching logic within a method. Here are some examples: `beq`: Break to code label if equal `bgt`: Break to code label if greater than `ble`: Break to code label if less than or equal to `blt`: Break to code label if less than All the branch-centric opcodes require that you specify a CIL code label to jump to if the result of the test is true.
`call`	This CIL opcode is used to call a member on a given type.
`newarr`, `newobj`	These CIL opcodes allow you to allocate a new array or new object type into memory (respectively).

Load (ld) (push) specific opcodes

The next broad category of CIL opcodes (a subset of which is shown in Table 19-5) is used to load (push) arguments onto the virtual execution stack. Note how these load-specific opcodes take an `ld` (load) prefix.

Table 19-5. *The Primary Stack-Centric Opcodes of CIL*

Opcode	Meaning in Life
`ldarg` (with numerous variations)	Loads a method's argument onto the stack. In addition to the general `ldarg` (which works in conjunction with a given index that identifies the argument), there are numerous other variations. For example, `ldarg` opcodes that have a numerical suffix (`ldarg.0`) hard-code which argument to load. As well, variations of the `ldarg` opcode allow you to hard-code the data type using the CIL constant notation shown in Table 19-4 (`ldarg_I4`, for an `int32`), as well as the data type and value (`ldarg_I4_5`, to load an `int32` with the value of `5`).
`ldc` (with numerous variations)	Loads a constant value onto the stack.
`ldfld` (with numerous variations)	Loads the value of an instance-level field onto the stack.
`ldloc` (with numerous variations)	Loads the value of a local variable onto the stack.
`ldobj`	Obtains all the values gathered by a heap-based object and places them on the stack.
`ldstr`	Loads a string value onto the stack.

Pop centric

Table 19-6. *Various Pop-Centric Opcodes*

Opcode	Meaning in Life
`pop`	Removes the value currently on top of the evaluation stack but does not bother to store the value
`starg`	Stores the value on top of the stack into the method argument at a specified index
`stloc` (with numerous variations)	Pops the current value from the top of the evaluation stack and stores it in a local variable list at a specified index
`stobj`	Copies a value of a specified type from the evaluation stack into a supplied memory address
`stsfld`	Replaces the value of a static field with a value from the evaluation stack

The maxstack directive

The .maxstack Directive

When you write method implementations using raw CIL, you need to be mindful of a special directive named `.maxstack`. As its name suggests, `.maxstack` establishes the maximum number of variables that may be pushed onto the stack at any given time during the execution of the method. The good news is that the `.maxstack` directive has a default value (8), which should be safe for a vast majority of methods you might be authoring. However, if you want to be explicit, you are able to manually calculate the number of local variables on the stack and define this value explicitly, like so:

```
.method public hidebysig instance void
  Speak() cil managed
{
  // During the scope of this method, exactly
  // 1 value (the string literal) is on the stack.
  .maxstack 1
  ldstr "Hello there..."
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

Local variables

```
.method public hidebysig static void
  MyLocalVariables() cil managed
{
  .maxstack 8
  // Define three local variables.
  .locals init (string myStr, int32 myInt, object myObj)
  // Load a string onto the virtual execution stack.
  ldstr "CIL code is fun!"
  // Pop off current value and store in local variable [0].
  stloc.0

  // Load a constant of type "i4"
  // (shorthand for int32) set to the value 33.
  ldc.i4.s 33
  // Pop off current value and store in local variable [1].
  stloc.1

  // Create a new object and place on stack.
  newobj instance void [mscorlib]System.Object::.ctor()
  // Pop off current value and store in local variable [2].
  stloc.2
  ret
}
```

Mapping Parameters to Local Variables in CIL

```
public static int Add(int a, int b)
{
  return a + b;
}
```

This innocent-looking method has a lot to say in terms of CIL. First, the incoming arguments (a and b) must be pushed onto the virtual execution stack using the ldarg (load argument) opcode. Next, the add opcode will be used to pop the next two values off the stack and find the summation and store the value on the stack yet again. Finally, this sum is popped off the stack and returned to the caller via the ret opcode. If you were to disassemble this C# method using ildasm.exe, you would find numerous additional tokens injected by the build process, but the crux of the CIL code is quite simple.

```
.method public hidebysig static int32 Add(int32 a,
  int32 b) cil managed
{
  .maxstack 2
  ldarg.0 // Load "a" onto the stack.
  ldarg.1 // Load "b" onto the stack.
  add     // Add both values.
  ret
}
```

The hidden this ref

The Hidden this Reference

Notice that the two incoming arguments (a and b) are referenced within the CIL code using their indexed position (index 0 and index 1), given that the virtual execution stack begins indexing at position 0.

One thing to be mindful of when you are examining or authoring CIL code is that every nonstatic method that takes incoming arguments automatically receives an implicit additional parameter, which is a reference to the current object (similar to the C# this keyword). Given this, if the Add() method were defined as *nonstatic*, like so:

```
// No longer static!
public int Add(int a, int b)
{
  return a + b;
}
```

the incoming a and b arguments are loaded using ldarg.1 and ldarg.2 (rather than the expected ldarg.0 and ldarg.1 opcodes). Again, the reason is that slot 0 actually contains the implicit this reference. Consider the following pseudocode:

```
// This is JUST pseudocode!
.method public hidebysig static int32 AddTwoIntParams(
  MyClass_HiddenThisPointer this, int32 a, int32 b) cil managed
```

Building .NET assemblies with CIL

- See examples on a DLL and user application
- page 678- 681

Dynamic Assemblies

- Hard work
- BUT GIVES REAL DYNAMIC POSSIBLITIES
- Is created in memory ”on the fly” and can be stored on disk for future use
- Example: **User defined business logic and user defined applications!!!**
- ”Bake” an assembly on run-time!

- Basic namespace : System.Reflection.Emit

System.Reflection.Emit namespace

- Contains many interesting classes
  - for instance
    - TypeBuilder
    - ILGenerator
    - OpCodes
- See table 19-7:

Table 18-8. *Select Members of the* System.Reflection.Emit *Namespace*

Members	Meaning in Life
AssemblyBuilder	Used to create an assembly (*.dll or *.exe) at runtime. *.exes must call the ModuleBuilder.SetEntryPoint() method to set the method that is the entry point to the module. If no entry point is specified, a *.dll will be generated.
ModuleBuilder	Used to define the set of modules within the current assembly.
EnumBuilder	Used to create a .NET enumeration type.
TypeBuilder	May be used to create classes, interfaces, structures, and delegates within a module at runtime.
MethodBuilder LocalBuilder PropertyBuilder FieldBuilder ConstructorBuilder CustomAttributeBuilder ParameterBuilder EventBuilder	Used to create type members (such as methods, local variables, properties, constructors, and attributes) at runtime.
ILGenerator	Emits CIL opcodes into a given type member.
OpCodes	Provides numerous fields that map to CIL opcodes. This type is used in conjunction with the various members of System.Reflection.Emit.ILGenerator.

System.Reflection.Emit.ILGenerator

- Has many methods that can emittere the structure in CIL
- See table 19-8 (This is only parts of it)

Table 18-9. *Various Methods of* ILGenerator

Method	Meaning in Life
BeginCatchBlock()	Begins a catch block
BeginExceptionBlock()	Begins an exception block for a nonfiltered exception
BeginFinallyBlock()	Begins a finally block
BeginScope()	Begins a lexical scope
DeclareLocal()	Declares a local variable
DefineLabel()	Declares a new label
Emit()	Is overloaded numerous times to allow you to emit CIL opcodes
EmitCall()	Pushes a call or callvirt opcode into the CIL stream
EmitWriteLine()	Emits a call to Console.WriteLine() with different types of values
EndExceptionBlock()	Ends an exception block
EndScope()	Ends a lexical scope
ThrowException()	Emits an instruction to throw an exception
UsingNamespace()	Specifies the namespace to be used in evaluating locals and watches for the current active lexical scope

AssemblyBuilderAccess

```
...

// Establish general assembly characteristics
// and gain access to the AssemblyBuilder type.
public static void CreateMyAsm(AppDomain curAppDomain)
{
AssemblyName assemblyName = new AssemblyName();
assemblyName.Name = "MyAssembly";
assemblyName.Version = new Version("1.0.0.0");
// Create new assembly within the current AppDomain.
AssemblyBuilder assembly =
curAppDomain.DefineDynamicAssembly(assemblyName,
AssemblyBuilderAccess.Save);
...
```

- In current AppDomain

AssemblyBuilderAccess enum

Table 18-10. *Common Values of the AssemblyBuilderAccess Enumeration*

Value	Meaning in Life
ReflectionOnly	Represents that a dynamic assembly can only be reflected over
Run	Represents that a dynamic assembly can be executed in memory but not saved to disk
RunAndSave	Represents that a dynamic assembly can be executed in memory and saved to disk
Save	Represents that a dynamic assembly can be saved to disk but not executed in memory

Example 1 : Dynamic assemblies (.NET 4.5)

- Simple example

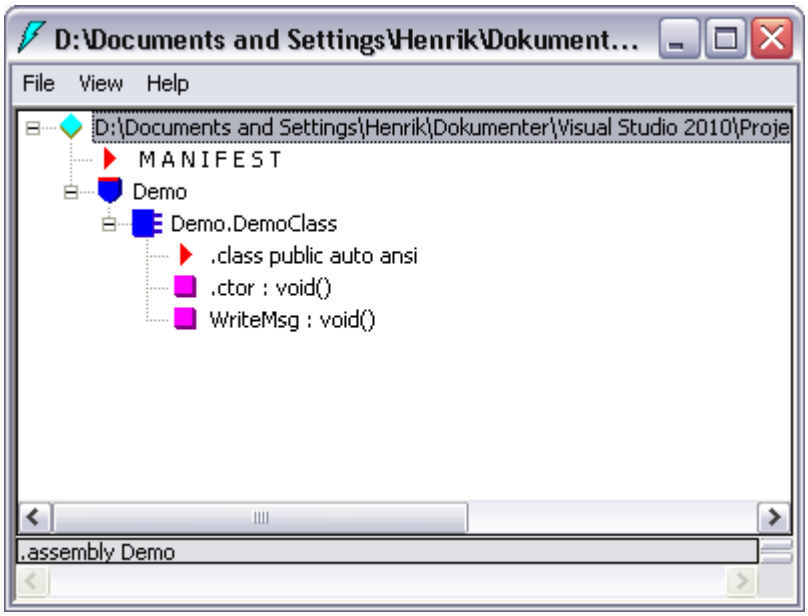
C# code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;
using System.Reflection.Emit;
using System.Threading;

namespace AssemblyBaker
{
    class Program
    {
        static void Main(string[] args)
        {
            AppDomain theDefault = Thread.GetDomain();
            Program p = new Program();
            p.Baker(theDefault);
            Console.WriteLine("Assembly constructed..");
            Assembly asm = Assembly.Load("Demo");
            Type t = asm.GetType("Demo.DemoClass");
            Object o = Activator.CreateInstance(t);
            MethodInfo mi = t.GetMethod("WriteMsg");
            mi.Invoke(o, null);

        }

        public void Baker(AppDomain curApp)
        {
            // general info
            AssemblyName asmName = new AssemblyName();
```

```
            asmName.Name = "Demo";
            asmName.Version = new Version("1.0.0.0");

            // create Assembly

            AssemblyBuilder assembly = curApp.DefineDynamicAssembly(asmName,

                                                AssemblyBuilderAccess.Save);

            // create a module in a single-file assembly

            ModuleBuilder module = assembly.DefineDynamicModule("Demo", "Demo.dll");
            TypeBuilder myType = module.DefineType("Demo.DemoClass", TypeAttributes.Public);
            myType.DefineDefaultConstructor(MethodAttributes.Public);

            MethodBuilder infoWriter = myType.DefineMethod("WriteMsg",

                                    MethodAttributes.Public,null, null);

            ILGenerator methodIl = infoWriter.GetILGenerator();

            methodIl.EmitWriteLine("Hi from dynamic assembly");
            methodIl.Emit(OpCodes.Ret);
             // create type and save
            myType.CreateType();
            assembly.Save("Demo.dll");

        }


    }
}
```

- A DLL (assembly) with the name (Demo.dll)
- with a default constructor
- and one method (writing a text)
- Dll is loaded dynamically
- Method is fetched and called

Ildasm used on Demo.dll



Constructor:

```
.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  ret
} // end of method DemoClass::.ctor
```

The Method WriteMsg

```
.method public instance void  WriteMsg() cil managed
{
  // Code size       11 (0xb)
  .maxstack  1
  IL_0000:  ldstr      "Hi from dynamic assembly"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  ret
} // end of method DemoClass::WriteMsg
```

Manifest

```
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         //
.z\V.4..
  .ver 4:0:0:0
}
.assembly Demo
{
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module Demo
// MVID: {92B74068-8FDA-423C-8517-3A6E8319FA8A}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x04040000
```

Example 2 : Dynamic assemblies (.NET 4.5)

- A simple calculator

C# Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;
using System.Reflection.Emit;
using System.Threading;

namespace AssemblyBaker
{
    class Program
    {
        static void Main(string[] args)
        {
            AppDomain theDefault = Thread.GetDomain();
            Program p = new Program();
            p.AssemblyBaker(theDefault);
            Console.WriteLine("Assembly constructed..");
```

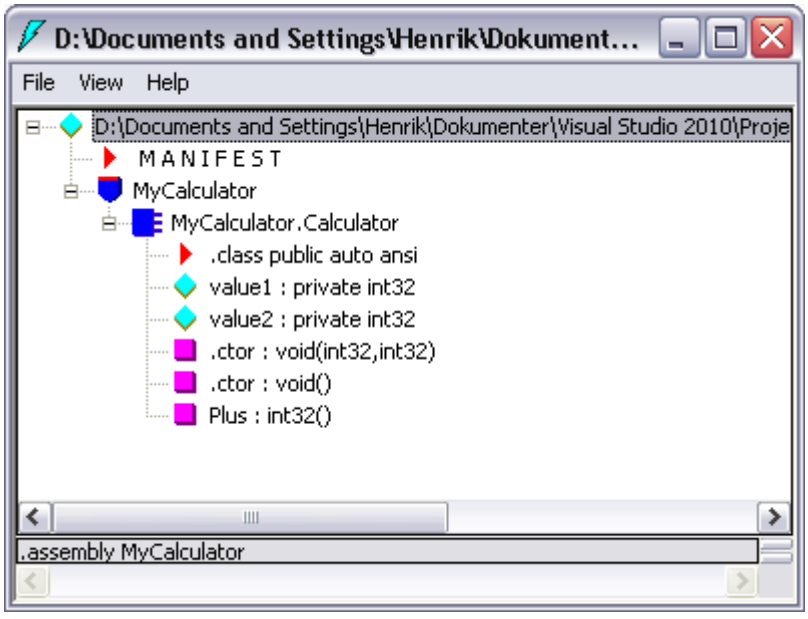
```
        object[] argum = new object[2];
        argum[0] = 23;
        argum[1] = 44;

        Assembly asm = Assembly.Load("MyCalculator");
        Type t = asm.GetType("MyCalculator.Calculator");
        Object o = Activator.CreateInstance(t,argum);
        MethodInfo mi = t.GetMethod("Plus");
        object obRes =  mi.Invoke(o, null);
        int result = (int)obRes;
        Console.WriteLine("RESULT {0}", result);



    }

     public void AssemblyBaker(AppDomain curApp)
    {
        // general info
        AssemblyName asmName = new AssemblyName();
        asmName.Name = "MyCalculator";
        asmName.Version = new Version("1.0.0.0");

        // Create Assembly

        AssemblyBuilder assembly =

                curApp.DefineDynamicAssembly(asmName,AssemblyBuilderAccess.Save);

        // Create module in a single-file assembly


        ModuleBuilder module = assembly.DefineDynamicModule("MyCalculator",

                                                            "MyCalculator.dll");

        // Define a public class

        TypeBuilder calc =

                    module.DefineType("MyCalculator.Calculator",TypeAttributes.Public);


        // now : content

        // two private Int32



        FieldBuilder value1 = calc.DefineField("value1", Type.GetType("System.Int32"),

                                                          FieldAttributes.Private);
        FieldBuilder value2 = calc.DefineField("value2", Type.GetType("System.Int32"),

                                                           FieldAttributes.Private);
```

```
            // Constructors

            /*****************************/

            calc.DefineDefaultConstructor(MethodAttributes.Public);

            /*****************************/

            Type objType = Type.GetType("System.Object");

            Type[] ctorParams = { typeof(int), typeof(int) };

            ConstructorBuilder calcCtor =
               calc.DefineConstructor(MethodAttributes.Public,
                                      CallingConventions.Standard, ctorParams);
            ILGenerator ctorIL = calcCtor.GetILGenerator();


            ctorIL.Emit(OpCodes.Ldarg_0);

            Type objClass = typeof(object);

            ConstructorInfo superConstructor = objClass.GetConstructor(new Type[0]);
            ctorIL.Emit(OpCodes.Call, superConstructor);
            ctorIL.Emit(OpCodes.Nop);
            ctorIL.Emit(OpCodes.Nop);
            ctorIL.Emit(OpCodes.Ldarg_0);
            ctorIL.Emit(OpCodes.Ldarg_1);
            ctorIL.Emit(OpCodes.Stfld, value1);
            ctorIL.Emit(OpCodes.Ldarg_0);
            ctorIL.Emit(OpCodes.Ldarg_2);
            ctorIL.Emit(OpCodes.Stfld, value2);
            ctorIL.Emit(OpCodes.Nop);
            ctorIL.Emit(OpCodes.Ret);



            /*****************************/


            // Define a Plus method

            /*****************************/

            MethodBuilder square = calc.DefineMethod("Plus", MethodAttributes.Public,
                                                     typeof(System.Int32), null);

            ILGenerator methodIl = square.GetILGenerator();
            methodIl.DeclareLocal(Type.GetType("System.Int32"));
            methodIl.Emit(OpCodes.Ldarg_0);
            methodIl.Emit(OpCodes.Ldfld,value1);
            methodIl.Emit(OpCodes.Ldarg_0);
            methodIl.Emit(OpCodes.Ldfld,value2);
            methodIl.Emit(OpCodes.Add);
            methodIl.Emit(OpCodes.Stloc_0);
            methodIl.Emit(OpCodes.Ldloc_0);
            methodIl.Emit(OpCodes.Ret);

            /*****************************/
```

```
            // Create the type and save

            calc.CreateType();

            assembly.Save("MyCalculator.dll");


        }
    }
}
```

- A class with two constructors
  - Call explicit SuperConstruktor if `calc.DefineDefaultConstructor(MethodAttributes.Public);` is not used
  - Constructor takes two ints as arguments
  - The Plus method adds the two private ints and return value

Ildasm on MyCalcultor.dll



The Default constructor

```
.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  ret
```

```
} // end of method Calculator::.ctor
```

Constructor with two arguments

```
.method public specialname rtspecialname
        instance void  .ctor(int32 A_1,
                             int32 A_2) cil managed
{
  // Code size       24 (0x18)
  .maxstack  4
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  nop
  IL_0007:  nop
  IL_0008:  ldarg.0
  IL_0009:  ldarg.1
  IL_000a:  stfld      int32 MyCalculator.Calculator::value1
  IL_000f:  ldarg.0
  IL_0010:  ldarg.2
  IL_0011:  stfld      int32 MyCalculator.Calculator::value2
  IL_0016:  nop
  IL_0017:  ret
} // end of method Calculator::.ctor
```

Plus method

```
.method public instance int32  Plus() cil managed
{
  // Code size       16 (0x10)
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:  ldarg.0
  IL_0001:  ldfld      int32 MyCalculator.Calculator::value1
  IL_0006:  ldarg.0
  IL_0007:  ldfld      int32 MyCalculator.Calculator::value2
  IL_000c:  add
  IL_000d:  stloc.0
  IL_000e:  ldloc.0
  IL_000f:  ret
} // end of method Calculator::Plus
```

Manifest

```
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         //
.z\V.4..
  .ver 4:0:0:0
}
```

```
.assembly MyCalculator
{
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module MyCalculator
// MVID: {AD12B185-FFE5-418E-B8B6-BBEE9F93ED39}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x03D80000
```

Output:

ModuleBuilder

- Supports (builds) the language defined expected types
  - Classes
  - Interfaces
  - Structs
  - and so on...

  - See table 19-10

Table 18-11. *Select Members of the ModuleBuilder Type*

Method	Meaning in Life
DefineEnum()	Used to emit a .NET enum definition
DefineResource()	Defines a managed embedded resource to be stored in this module
DefineType()	Constructs a TypeBuilder, which allows you to define value types, interfaces, and class types (including delegates)

TypeAttributes Enumeration

- Describes the format of the type
  - See table 19-11

Table 18-12. *Select Members of the TypeAttributes Enumeration*

Member	Meaning in Life
Abstract	Specifies that the type is abstract
Class	Specifies that the type is a class
Interface	Specifies that the type is an interface
NestedAssembly	Specifies that the class is nested with assembly visibility and is thus accessible only by methods within its assembly
NestedFamAndAssem	Specifies that the class is nested with assembly and family visibility, and is thus accessible only by methods lying in the intersection of its family and assembly
NestedFamily	Specifies that the class is nested with family visibility and is thus accessible only by methods within its own type and any subtypes
NestedFamORAssem	Specifies that the class is nested with family or assembly visibility, and is thus accessible only by methods lying in the union of its family and assembly
NestedPrivate	Specifies that the class is nested with private visibility
NestedPublic	Specifies that the class is nested with public visibility
NotPublic	Specifies that the class is not public
Public	Specifies that the class is public
Sealed	Specifies that the class is concrete and cannot be extended
Serializable	Specifies that the class can be serialized

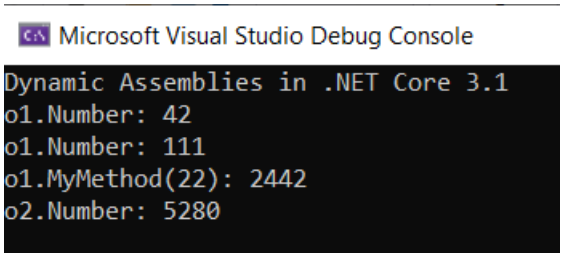
Example 3 in .NET Core 3.1

```
using System;
using System.Reflection.Emit;
using System.Reflection;
using System.Threading;

namespace TestApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Dynamic Assemblies in .NET Core 3.1");
         /*   Creates this class and use it
        public class MyDynamicType
        {
            private int m_number;

            public MyDynamicType() : this(42) {}
            public MyDynamicType(int initNumber)
            {
                m_number = initNumber;
            }

            public int Number
            {
                get { return m_number; }
                set { m_number = value; }
            }

            public int MyMethod(int multiplier)
            {
                return m_number * multiplier;
            }
        }
        */
            AssemblyBaker();


        }
        public static void AssemblyBaker()
        {

                AssemblyName aName = new AssemblyName("DynamicAssemblyExample");
                var ab =
                    AssemblyBuilder.DefineDynamicAssembly(
                        aName,
                        AssemblyBuilderAccess.Run);

                // For a single-module assembly, the module name is usually
                // the assembly name plus an extension.
                ModuleBuilder mb =
                    ab.DefineDynamicModule( aName.Name + ".dll");

                TypeBuilder tb = mb.DefineType(
                    "MyDynamicType",
                     TypeAttributes.Public);
```

```
            // Add a private field of type int (Int32).
            FieldBuilder fbNumber = tb.DefineField(
                "m_number",
                typeof(int),
                FieldAttributes.Private);

            // Define a constructor that takes an integer argument and
            // stores it in the private field.
            Type[] parameterTypes = { typeof(int) };
            ConstructorBuilder ctor1 = tb.DefineConstructor(
                MethodAttributes.Public,
                CallingConventions.Standard,
                parameterTypes);

            ILGenerator ctor1IL = ctor1.GetILGenerator();
            // For a constructor, argument zero is a reference to the new
            // instance. Push it on the stack before calling the base
            // class constructor. Specify the default constructor of the
            // base class (System.Object) by passing an empty array of
            // types (Type.EmptyTypes) to GetConstructor.
            ctor1IL.Emit(OpCodes.Ldarg_0);
            ctor1IL.Emit(OpCodes.Call,
                typeof(object).GetConstructor(Type.EmptyTypes));
            // Push the instance on the stack before pushing the argument
            // that is to be assigned to the private field m_number.
            ctor1IL.Emit(OpCodes.Ldarg_0);
            ctor1IL.Emit(OpCodes.Ldarg_1);
            ctor1IL.Emit(OpCodes.Stfld, fbNumber);
            ctor1IL.Emit(OpCodes.Ret);

            // Define a default constructor that supplies a default value
            // for the private field. For parameter types, pass the empty
            // array of types or pass null.
            ConstructorBuilder ctor0 = tb.DefineConstructor(
                MethodAttributes.Public,
                CallingConventions.Standard,
                Type.EmptyTypes);

            ILGenerator ctor0IL = ctor0.GetILGenerator();
            // For a constructor, argument zero is a reference to the new
            // instance. Push it on the stack before pushing the default
            // value on the stack, then call constructor ctor1.
            ctor0IL.Emit(OpCodes.Ldarg_0);
            ctor0IL.Emit(OpCodes.Ldc_I4_S, 42);
            ctor0IL.Emit(OpCodes.Call, ctor1);
            ctor0IL.Emit(OpCodes.Ret);

            // Define a property named Number that gets and sets the private
            // field.
            //
            // The last argument of DefineProperty is null, because the
            // property has no parameters. (If you don't specify null, you must
            // specify an array of Type objects. For a parameterless property,
            // use the built-in array with no elements: Type.EmptyTypes)
            PropertyBuilder pbNumber = tb.DefineProperty(
                "Number",
                PropertyAttributes.HasDefault,
                typeof(int),
                null);
```

```
            // The property "set" and property "get" methods require a special
            // set of attributes.
            MethodAttributes getSetAttr = MethodAttributes.Public |
                MethodAttributes.SpecialName | MethodAttributes.HideBySig;

            // Define the "get" accessor method for Number. The method returns
            // an integer and has no arguments. (Note that null could be
            // used instead of Types.EmptyTypes)
            MethodBuilder mbNumberGetAccessor = tb.DefineMethod(
                "get_Number",
                getSetAttr,
                typeof(int),
                Type.EmptyTypes);

            ILGenerator numberGetIL = mbNumberGetAccessor.GetILGenerator();
            // For an instance property, argument zero is the instance. Load the
            // instance, then load the private field and return, leaving the
            // field value on the stack.
            numberGetIL.Emit(OpCodes.Ldarg_0);
            numberGetIL.Emit(OpCodes.Ldfld, fbNumber);
            numberGetIL.Emit(OpCodes.Ret);

            // Define the "set" accessor method for Number, which has no return
            // type and takes one argument of type int (Int32).
            MethodBuilder mbNumberSetAccessor = tb.DefineMethod(
                "set_Number",
                getSetAttr,
                null,
                new Type[] { typeof(int) });

            ILGenerator numberSetIL = mbNumberSetAccessor.GetILGenerator();
            // Load the instance and then the numeric argument, then store the
            // argument in the field.
            numberSetIL.Emit(OpCodes.Ldarg_0);
            numberSetIL.Emit(OpCodes.Ldarg_1);
            numberSetIL.Emit(OpCodes.Stfld, fbNumber);
            numberSetIL.Emit(OpCodes.Ret);

            // Last, map the "get" and "set" accessor methods to the
            // PropertyBuilder. The property is now complete.
            pbNumber.SetGetMethod(mbNumberGetAccessor);
            pbNumber.SetSetMethod(mbNumberSetAccessor);

            // Define a method that accepts an integer argument and returns
            // the product of that integer and the private field m_number. This
            // time, the array of parameter types is created on the fly.
            MethodBuilder meth = tb.DefineMethod(
                "MyMethod",
                MethodAttributes.Public,
                typeof(int),
                new Type[] { typeof(int) });

            ILGenerator methIL = meth.GetILGenerator();
            // To retrieve the private instance field, load the instance it
            // belongs to (argument zero). After loading the field, load the
            // argument one and then multiply. Return from the method with
            // the return value (the product of the two numbers) on the
            // execution stack.
            methIL.Emit(OpCodes.Ldarg_0);
            methIL.Emit(OpCodes.Ldfld, fbNumber);
```

```
                methIL.Emit(OpCodes.Ldarg_1);
                methIL.Emit(OpCodes.Mul);
                methIL.Emit(OpCodes.Ret);

                // Finish the type.
                Type t = tb.CreateType();



                // Because AssemblyBuilderAccess includes Run, the code can be
                // executed immediately. Start by getting reflection objects for
                // the method and the property.
                MethodInfo mi = t.GetMethod("MyMethod");
                PropertyInfo pi = t.GetProperty("Number");

                // Create an instance of MyDynamicType using the default
                // constructor.
                object o1 = Activator.CreateInstance(t);

                // Display the value of the property, then change it to 111 and
                // display it again. Use null to indicate that the property
                // has no index.
                Console.WriteLine("o1.Number: {0}", pi.GetValue(o1, null));
                pi.SetValue(o1, 111, null);
                Console.WriteLine("o1.Number: {0}", pi.GetValue(o1, null));

                // Call MyMethod, passing 22, and display the return value, 22
                // times 127. Arguments must be passed as an array, even when
                // there is only one.
                object[] arguments = { 22 };
                Console.WriteLine("o1.MyMethod(22): {0}",
                    mi.Invoke(o1, arguments));

                // Create an instance of MyDynamicType using the constructor
                // that specifies m_Number. The constructor is identified by
                // matching the types in the argument array. In this case,
                // the argument array is created on the fly. Display the
                // property value.
                object o2 = Activator.CreateInstance(t,
                    new object[] { 5280 });
                Console.WriteLine("o2.Number: {0}", pi.GetValue(o2, null));
            }
        }

        /* This code produces the following output:

        o1.Number: 42
        o1.Number: 127
        o1.MyMethod(22): 2794
        o2.Number: 5280
         */
    }
```


Microsoft Visual Studio Debug Console
Dynamic Assemblies in .NET Core 3.1
o1.Number: 42
o1.Number: 111
o1.MyMethod(22): 2442
o2.Number: 5280