

.NET Namespaces

Type Reflection – Late Binding

Attribute programming

Namespaces:

- can group logic
 - across files
 - import (using) of logic in DLL's

Example:

```
// Circle.cs
using System;
namespace MyShapes
{
    // Circle class
    public class Circle { /* Interesting methods... */ }
}
// Hexagon.cs
using System;
namespace MyShapes
{
    // Hexagon class
    public class Hexagon { /* More interesting methods... */ }
}
// Square.cs
using System;
namespace MyShapes
{
    // Square class
    public class Square { /* Even more interesting methods... */ }
}
```

Name clashes can be avoided with namespaces

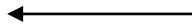
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Bus
{
    public class Motor
    {
    }
}
```

```
namespace Bil
{
    public class Motor
    {

    }
}

namespace Kap14
{
    using Bus;
    using Bil;
    class Program
    {
        static void Main(string[] args)
        {
            Bil.Motor motor = new Bil.Motor();
            Bus.Motor storMotor = new Bus.Motor();
        }
    }
}
```



- Use fully qualified names

The use of alias

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

namespace Bus
{
    public class Motor
    {

    }
}

namespace Bil
{
    public class Motor
    {

    }
}

namespace Kap14
{
    using Bus;
    using Bil;
```

```

using MinBusMotor = Bus.Motor;

class Program
{
    static void Main(string[] args)
    {
        Bil.Motor motor = new Bil.Motor();
        Bus.Motor storMotor = new Bus.Motor();
        MinBusMotor minAndenStoreMotor = new MinBusMotor(); ←
    }
}

```

Nested namespaces

```

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

namespace Bus
{
    public class Motor
    {

    }
}

namespace Bil
{
    public class Motor
    {

    }
}

namespace Dyr
{
    namespace Fugl
    {
        namespace Undulat
        {
            public class Pip
            {

            }
        }
    }
}

namespace Kap14
{
    using Bus;

```

```
using Bil;
using MinBusMotor = Bus.Motor;

class Program
{
    static void Main(string[] args)
    {
        Bil.Motor motor = new Bil.Motor();
        Bus.Motor storMotor = new Bus.Motor();
        MinBusMotor minAndenStoreMotor = new MinBusMotor();

        Dyr.Fugl.Undulat.Pip pipFugl = new Dyr.Fugl.Undulat.Pip(); ←
    }
}
```

- Of course also access using **using**

```
using Dyr.Fugl.Undulat;
...
Pip nyPipFugl = new Pip();
...
```

Note Be aware that overuse of C# aliases can result in a confusing codebase. If other programmers on your team are unaware of your custom aliases, they could assume the aliases refer to types in the .NET base class libraries and become quite confused when they can't find these tokens in the .NET Framework SDK documentation!

Default namespace

- Default namespace i Visual:

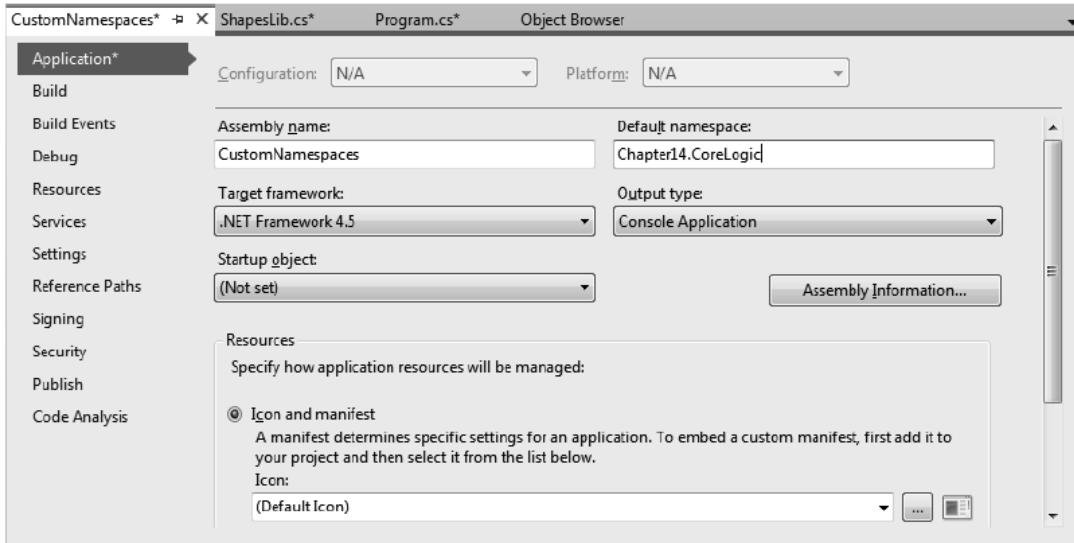


Figure 14-1. Configuring the default namespace

Note The Visual Studio property pages still refer to the root namespace as the “Default namespace”. You will see next why I refer to it as the “Root namespace”.

If not using Visual Studio (or even with Visual Studio), you can also configure the root namespace by updating the project (*.csproj) file. With .NET Core projects, editing the project file is as easy as double-clicking the project file in Solution Explorer (or right-clicking the project file in Solution Explorer and selecting “Edit project file”). Once the file is open, update the main PropertyGroup by adding in the RootNamespace node, like this:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <RootNamespace>CustomNamespaces2</RootNamespace>
  </PropertyGroup>

</Project>
```

.NET Core Assemblies

Code reuse

- code library
 - of the type DLL
 - Can in principle also be another .exe (this is not normal!)
- Language independent
 - Could be C# used in a MC++ application
 - or other language
 - Language independent code reuse

Design

- From simple stand-alone assemblies
- Reuse of business logic
- It is possible to have two unique namespaces containing the same types
 - They are regarded as unique because **assemblies also is a scope**

.NET has versioning

- <major>
- <minor>
- <build>
- <revision>
- AUTOMATICALLY
 - 1.0.0.0

Version number is together with a public key in the **strongly name** process!

.NET is self describing

- Manifest
- Metadata

CLR can read the content af an assembly. Thereby the Windows System Registry is not needed!

The content of .NET assemblies

- An executable consists of:
 - standard Windows header (OS header)
 - type of application (console /windows app ??)
 - CLR (Common Language Runtime) header
 - Identification of metadata
 - Version check on run-time
 - CIL (Common Intermediate Language) code
 - CIL compiled on run-time by the "Jitter"
 - Type metadata
 - Gets for instance information about WriteLine in the extern System.Console class, if used
 - Assembly manifest
 - documents every module in the assembly

Windows Header File

Dump of file carlibrary.dll

PE signature found

File Type: DLL

FILE HEADER VALUES

14C machine (x86)

3 number of sections

A6F7A030 time date stamp

0 file pointer to symbol table

0 number of symbols

E0 size of optional header

2022 characteristics

Executable

Application can handle large (>2GB) addresses

DLL

...

CLR File header

- Example

```
Dump of file CarLibrary.dll
```

File Type: DLL

clr Header:

```
48 cb
2.05 runtime version
2160 [ AB4] RVA [size] of MetaData Directory
  1 flags
    IL Only
  0 entry point token
  0 [ 0] RVA [size] of Resources Directory
  0 [ 0] RVA [size] of StrongNameSignature Directory
  0 [ 0] RVA [size] of CodeManagerTable Directory
  0 [ 0] RVA [size] of VTableFixups Directory
  0 [ 0] RVA [size] of ExportAddressTableJumps Directory
  0 [ 0] RVA [size] of ManagedNativeHeader Directory
```

Summary

```
2000 .reloc
2000 .rsrc
2000 .text
```

Mixing .NET Core and .NET standard

- Can be done - but

Each .NET Standard version defines a common set of APIs that must be supported by all .NET versions (.NET, .NET Core, Xamarin, etc.) to conform to the standard. For example, if you were to build a class library as a .NET Standard 2.0 project, it can be referenced by .NET 4.61+ and .NET Core 2.0+ (plus various versions of Xamarin, Mono, Universal Windows Platform, and Unity).

This means you could move the code from your .NET class libraries into .NET Standard 2.0 class libraries, and they can be shared by .NET Core and .NET applications. Much better than supporting duplicate copies of the same code, one for each framework.

If this sounds too good to be true, there is indeed a catch. Each .NET Standard version represents the lowest common denominator for the frameworks. That means the lower the version, the less that you can do in your class library. While .NET Core 3.1 can reference a .NET Standard 2.0 library, you cannot use a significant number of C# 8 in a .NET Standard 2.0 library. You must use .NET Standard 2.1 for full C# 8 support. And .NET 4.8 (the latest/last version of the original .NET framework) only goes up to .NET Standard 2.0.

Still a good mechanism for leveraging existing code in newer applications, but not a silver bullet.

CIL Code

- The class above is added a constructor and a method

```

namespace Dyr
{
    namespace Fugl
    {
        namespace Undulat
        {
            public class Pip
            {
                private int antalPip;

                public Pip(int antal)
                {
                    antalPip = antal;
                }

                public void PipSå()
                {
                    for (int i = 0; i < antalPip; i++)
                        Console.WriteLine("Pip");
                }
            }
        }
    }
}

...

//Creating and call

Dyr.Fugl.Undulat.Pip pipFugl = new Dyr.Fugl.Undulat.Pip(1);
    Pip nyPipFugl = new Pip(22);
    pipFugl.PipSå();
    nyPipFugl.PipSå();
...

```

- CIL Code: Constructor

```

.method public hidebysig specialname rtspecialname
           instance void  .ctor(int32 antal) cil managed
{
    // Code size      17 (0x11)
    .maxstack  8
    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 Dyr.Fugl.Undulat.Pip::antalPip
IL_000f: nop
IL_0010: ret
} // end of method Pip::ctor

```

- CIL Code: Method PipSå()

```

.method public hidebysig instance void  'PipSå'() cil managed
{
    // Code size      34 (0x22)
    .maxstack 2
    .locals init ([0] int32 i,
                 [1] bool CS$4$0000)
    IL_0000:  nop
    IL_0001:  ldc.i4.0
    IL_0002:  stloc.0
    IL_0003:  br.s      IL_0014
    IL_0005:  ldstr     "Pip"
    IL_000a:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000f:  nop
    IL_0010:  ldloc.0
    IL_0011:  ldc.i4.1
    IL_0012:  add
    IL_0013:  stloc.0
    IL_0014:  ldloc.0
    IL_0015:  ldarg.0
    IL_0016:  ldfld      int32 Dyr.Fugl.Undulat.Pip::antalPip
    IL_001b:  clt
    IL_001d:  stloc.1
    IL_001e:  ldloc.1
    IL_001f:  brtrue.s   IL_0005
    IL_0021:  ret
} // end of method Pip::'PipSå'

```

- For construction from line 5 to 1F
- Ends with **ret** (return)

Download cache – The idea of the .netmodule

- If the code is placed in a .netmodule it can be loaded when wanted
 - Instead of a complete assembly
 - Is logical related via the primary module manifest

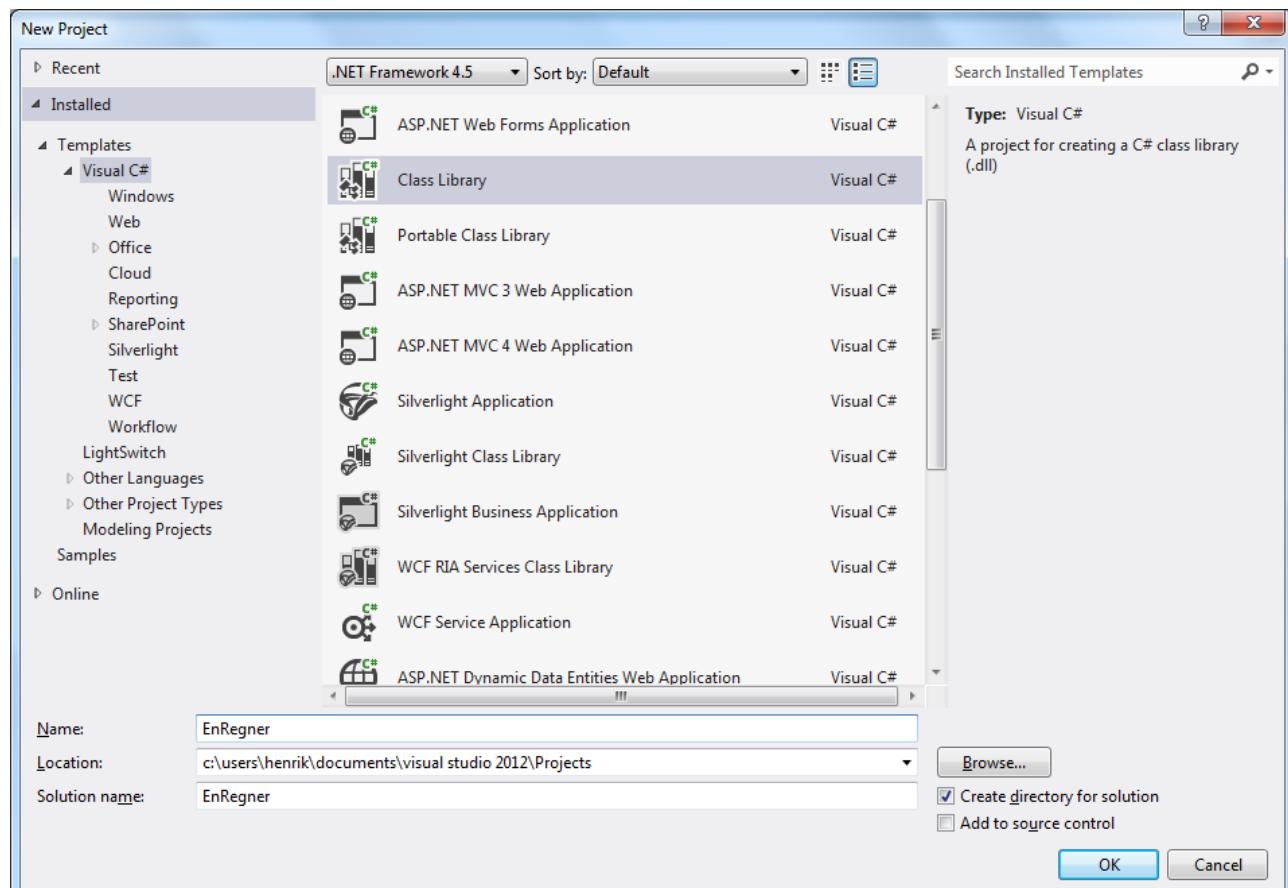
Example is shown later!

Assemblies

Code reuse: Using assemblies (creating DLL's)

- Some bigger applications
 - using own libraries

Example:



Content (just for demonstration) :

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

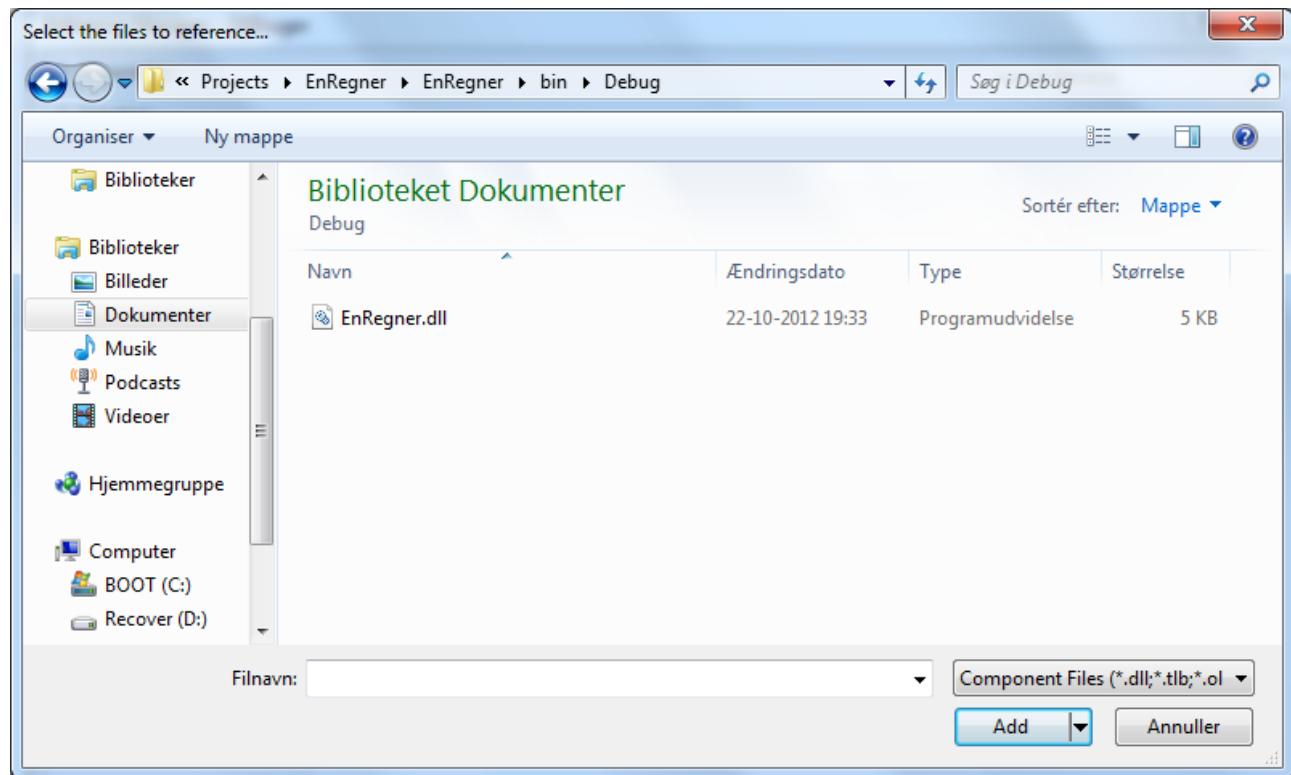
namespace EnRegner
{
    public class EnRegner
    {

        public double Plus(double a, double b)
        {
            return a + b;
        }

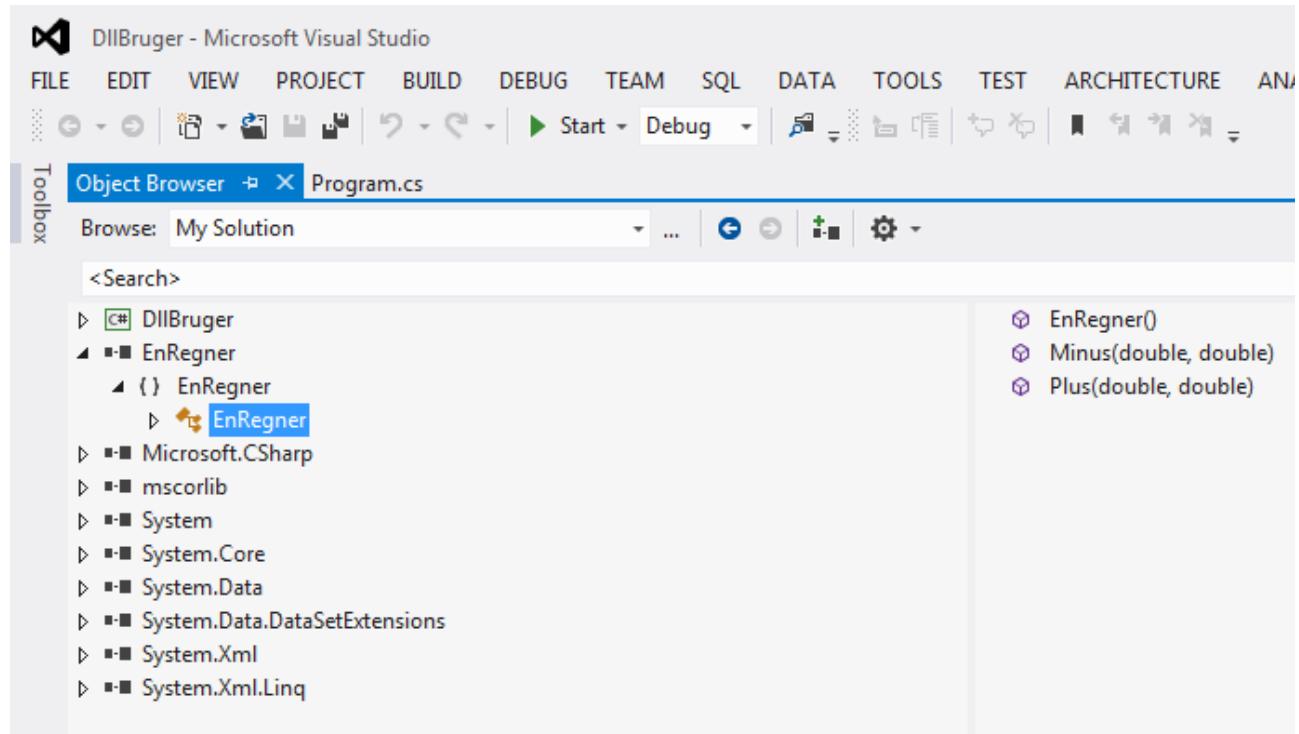
        public double Minus(double a, double b)
        {
            return a - b;
        }
    }
}
```

Make a "User Project"

- "Add Reference" to Library project (Project | Add Reference)



Now the content of the library can be seen (and used):



Using a DLL in a project

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

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

            EnRegner.EnRegner myCalc = new EnRegner();

            double res = myCalc.Plus(12.8, 44.9);
            Console.WriteLine(res);
            res = myCalc.Minus(45.8, 44.9);
            Console.WriteLine(res);
        }
    }
}
```

Configuring Applications

While it is possible to keep all of the information needed by your .NET Core application in the source code, being able to change certain values at runtime is vitally important in most applications of significance. This is typically done through a configuration file that is shipped with your application.

Note The previous .NET framework relied mostly on XML files named app.config (or web.config for ASP.NET applications). While XML-based configuration files can still be used, the main method for configuring .NET Core applications is with JSON (JavaScript Object Notation) files as shown in this section. Configuration will be addressed in depth in the chapters in WPF and ASP.NET Core.

To illustrate the process, create a new .NET Core Console application named FunWithConfiguration. Add the following package reference to your project:

```
dotnet add package Microsoft.Extensions.Configuration.Json
```

This adds a reference for the JSON file-based .NET Core configuration subsystem (and its dependencies) into your project. To leverage this, first add a new JSON file into your project named appsettings.json. Update the project file to make sure the file is always copied to the output directory when the project is built:

```
<ItemGroup>
  <None Update="appsettings.json">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

Finally, update the file to match the following:

```
{
  "CarName": "Suzy"
}
```

Next

Update the Main method to the following:

```
static void Main(string[] args)
{
    IConfiguration config = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json", true, true)
        .Build();
}
```

The new configuration system starts with a ConfigurationBuilder. This allows you to add multiple files, set properties (such as where the configuration files are located), and then finally build the configuration into an instance of IConfiguration.

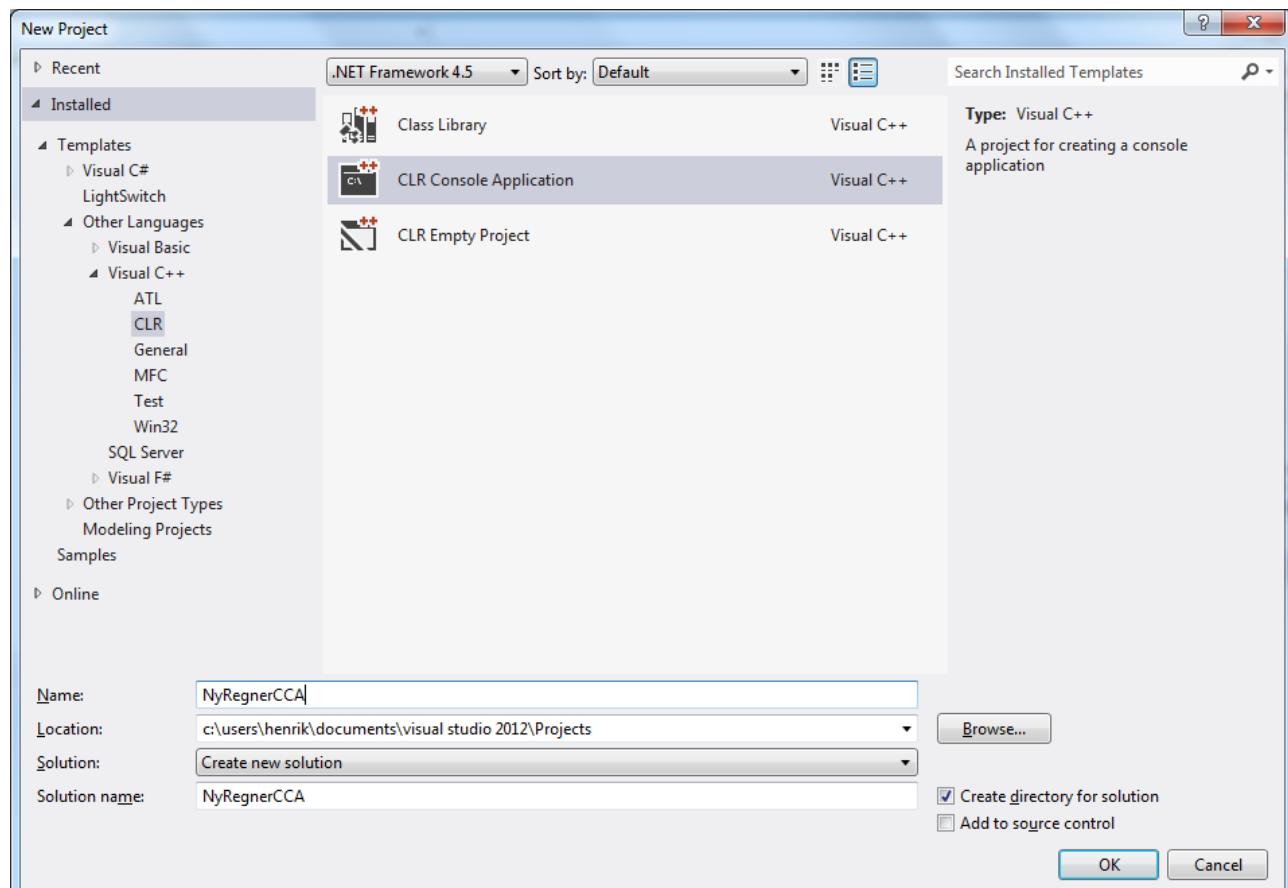
Once you have an instance of IConfiguration, you call it much the same way as in .NET 4.8. Add the following to the bottom of the Main method, and when you run the app, you will see the value written to the console:

```
Console.WriteLine($"My car's name is {config["CarName"]}");
Console.ReadLine();
```

In addition to JSON files, there are configuration packages for supporting environment variables, Azure Key Vault, command-line arguments, and many more. Reference the .NET Core documentation for more information.

Mix C# with C++

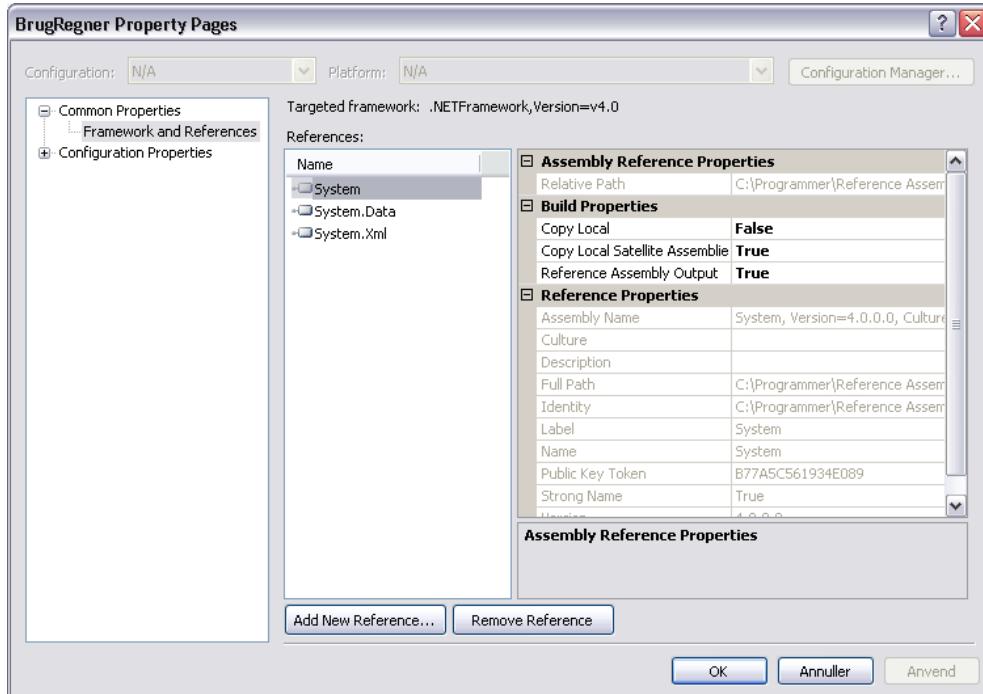
- The DLL from before is reused
- Remember: still "Project | Add Reference"



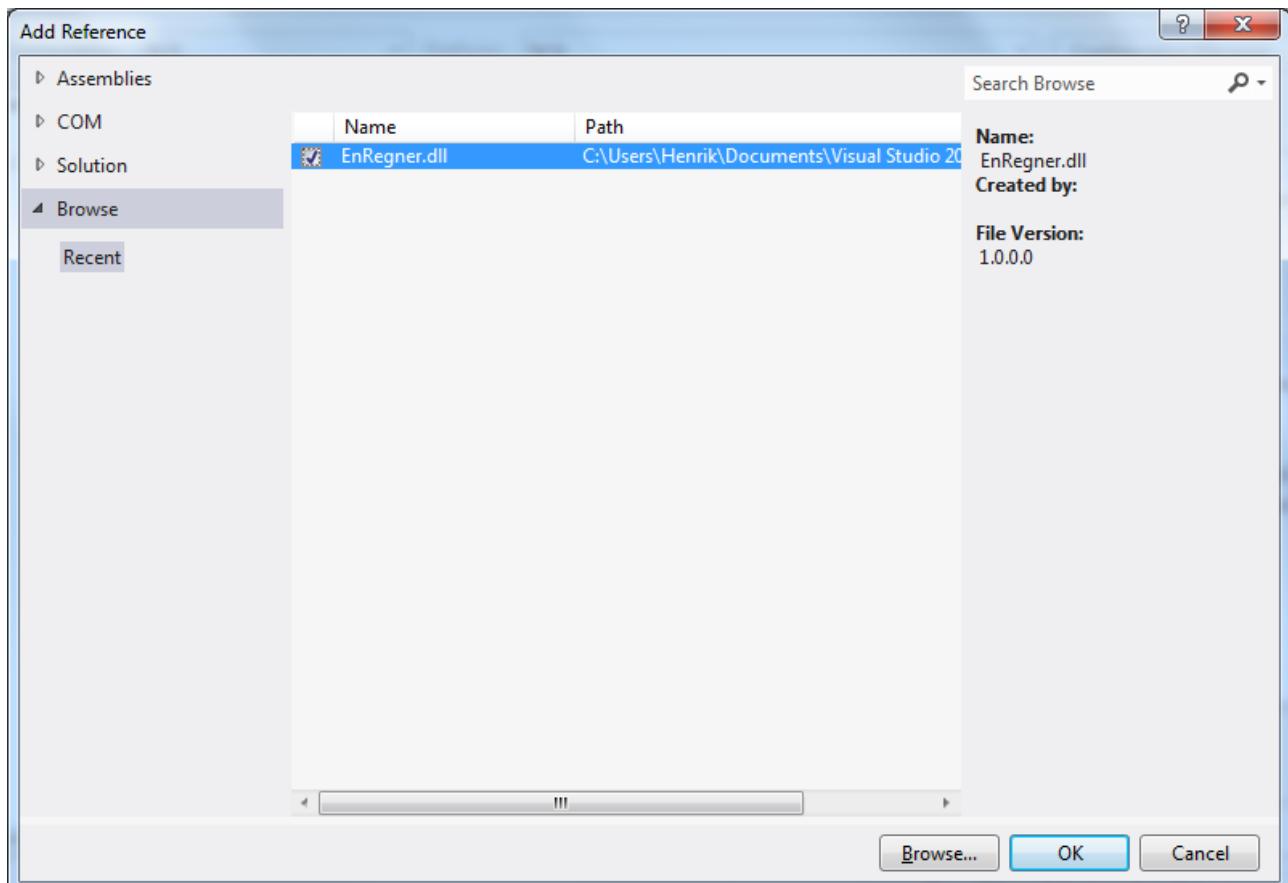
Using the DLL in C++ code

Example: using the DLL above:

- right-click on the project (in Solution Explorer)
- Choose: References



- Via properties: "Add New Reference"



- MC++ Code

```
// NyRegnerCCA.cpp : main project file.

#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
    EnRegner::EnRegner ^pEnRegner = gcnew EnRegner::EnRegner();
    double res   = pEnRegner->Plus(23.9,66.8);
    Console::WriteLine(res);
    res   = pEnRegner->Minus(286.9,56.2);
    Console::WriteLine(res);

    Console::ReadLine();

    return 0;
}
```

- Notice the notation and the gcnew keyword
 - gcnew = new without use of delete (Garbage Collected)



Closer look on assemblies

AssemblyInfo

Figure 16-4. Editing assembly information using Visual Studio's Properties window

Another way to add the metadata to your assembly is directly in the *.csproj project file. The following update to the main PropertyGroup in the project file does the same thing as filling in the form shown in Figure 16-4.

```
<PropertyGroup>
<TargetFramework>netcoreapp3.1</TargetFramework>
<Copyright>Copyright 2017</Copyright>
<Authors>Phil Japikse</Authors>
<Company>Apress</Company>
<Product>Pro C# 8</Product>
<PackageId>CarLibrary</PackageId>
<Description>This is an awesome library for cars.</Description>
<AssemblyVersion>1.0.0.1</AssemblyVersion>
<FileVersion>1.0.0.2</FileVersion>
<Version>1.0.0.3</Version>
</PropertyGroup>
```

Exploring the CIL

Recall that an assembly does not contain platform-specific instructions; rather, it contains platform-agnostic Common Intermediate Language (CIL) instructions. When the .NET Core runtime loads an assembly into memory, the underlying CIL is compiled (using the JIT compiler) into instructions that can be understood by the target platform. For example, the TurboBoost() method of the SportsCar class is represented by the following CIL:

```
.method public hidebysig virtual instance void  TurboBoost() cil managed
{
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr "Ramming speed! Faster is better..."
    IL_0006:  call  void [System.Console]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
}
// end of method SportsCar::TurboBoost
```

Exploring the Type Metadata

Before you build some applications that use your custom .NET library, examine the metadata for the types within the `CarLibrary.dll` assembly. For an example, here is the `TypeDef` for the `EngineState` enum:

```
TypeDef #1 (02000002)
-----
TypeDefName: CarLibrary.EngineState (02000002)
Flags      : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass] (00000101)
Extends   : 0100000E [TypeRef] System.Enum
Field #1 (04000001)

-----
Field Name: value_ (04000001)
Flags      : [Public] [SpecialName] [RTSpecialName] (00000606)
CallCnvtn: [FIELD]
Field type: I4

Field #2 (04000002)
-----
```

Building a C# Client Application

Because each of the `CarLibrary` types has been declared using the `public` keyword, other .NET Core applications are able to use them as well. Recall that you may also define types using the `C# internal` keyword (in fact, this is the default C# access mode). Internal types can be used only by the assembly in which they are defined. External clients can neither see nor create types marked with the `internal` keyword.

Note The exception to this rule is when an assembly explicitly allows access to another assembly using the `InternalsVisibleTo` attribute, covered shortly.

To use your library's functionality, create a new C# Console Application project named `CSharpCarClient` in the same solution as the `CarLibrary`. Again, you can do this using Visual Studio (right-click the solution, select Add ➤ New Project) or using the command line (three lines, each executed separately).

```
dotnet new console -n CSharpCarClient -f netcoreapp3.1
dotnet add CSharpCarClient reference CarLibrary
dotnet sln add CSharpCarClient
```

```
using System;
// Don't forget to import the CarLibrary namespace!
using CarLibrary;

namespace CSharpCarClient
{
    public class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** C# CarLibrary Client App *****");
            // Make a sports car.
            SportsCar viper = new SportsCar("Viper", 240, 40);
            viper.TurboBoost();

            // Make a minivan.
            MiniVan mv = new MiniVan();
            mv.TurboBoost();

            Console.WriteLine("Done. Press any key to terminate");
            Console.ReadLine();
        }
    }
}
```

Exposing internal Types to Other Assemblies

As mentioned earlier, [internal classes are only visible to other objects in the project where they defined](#). The sidebar note teased about an exception to this, and now is the time to investigate this.

Begin by adding a new class named `MyInternalClass` to the `CarLibrary` project, and update the code to the following:

```
namespace CarLibrary
{
    internal class MyInternalClass
    {
    }
}
```

Note Why expose internal types at all? This is usually done for unit and integration testing. Developers want to be able to test their code but not necessarily expose it beyond the borders of the assembly.

Using an Assembly Attribute

Chapter 17 will cover attributes in depth, but for now open up the Car.cs class in the CarLibrary project, and add the following attribute and using statement:

```
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("CSharpCarClient")]
namespace CarLibrary
{
}
```

The `InternalsVisibleTo` attribute takes the name of the project that is allowed to see into the class that has the attribute set. Note that other projects cannot “ask” for this permission; it has to be granted by the project holding the internal types.

Note Previous versions of .NET leveraged the `AssemblyInfo.cs` class, which still exists in .NET Core, but is autogenerated and not meant for developer consumption.

Now you can update the CSharpCarClient project by adding the following code to the Main method:

```
var internalClassInstance = new MyInternalClass();
```

This works perfectly. Now try to do the same thing in the VisualBasicCarClient Main method:

```
'Will not compile
'Dim internalClassInstance = New MyInternalClass()
```

Because the VisualBasicCarClient library was not granted permission to see the internals, the previous line of code will not compile.

NuGet and .NET Core

NuGet is the package manager for .NET and .NET Core. It is a mechanism to share software in a format that .NET Core applications understand and is the default mechanism for loading .NET Core and its related framework pieces (ASP.NET Core, EF Core, etc.). Many organizations package their standard assemblies for cross-cutting concerns (like logging and error reporting) into NuGet packages for consumption into their line-of-business applications.

Packaging Assemblies with NuGet

To see this in action, we will turn the CarLibrary into a NuGet package and then reference it from the two client applications.

The NuGet Package properties can be accessed from the project's property pages. Right-click the CarLibrary project and select Properties. Navigate to the Package page, and see the values that we entered before to customize the assembly. There are additional properties that can be set for the NuGet package (i.e., license agreement acceptance, project information, such as URL and repository location).

For this example, we don't need to set any additional properties except to check the "Generate NuGet package on build" check box. This will cause the package to be rebuilt every time the software is built. By default, the package will be created in the bin/Debug or bin/Release folder, depending on which configuration is selected.

Packages can also be created from the command line, and the CLI provides more options than Visual Studio. For example, to build the package and place it in a directory called Publish, enter the following commands (in the CarLibrary project directory). The first command builds the assembly, and the second packages up the NuGet package.

```
dotnet build -c Release  
dotnet pack -o .\Publish -c Debug
```

Note Debug is the default configuration, so it's not necessary, but is included to make it clear what the intent is.

Reference

Referencing NuGet Packages

You might be wondering where the packages that were added in the previous examples actually came from. The location of NuGet packages is controlled by an XML-based file named NuGet.Config. On Windows, this file is located in the %appdata%\NuGet directory. This is the main file. Open it up, and you will see several package sources:

```
<?xml version="1.0" encoding="utf-8"?>  
<configuration>  
  <packageSources>  
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json" protocolVersion="3" />  
    <add key="Microsoft Visual Studio Offline Packages" value="C:\Program Files (x86)\  
      Microsoft SDKs\NuGetPackages\" />  
  </packageSources>  
</configuration>
```

The previous file listing shows two sources. The first points to NuGet.org, which is the largest NuGet package repository in the world. The second is on your local drive and is used by Visual Studio as a cache of packages.

The import item to note is that NuGet.Config files are *additive*. To add additional sources without changing the list for the entire system, you can add additional NuGet.Config files. Each file is valid for the directory that it's placed in as well as any subdirectory. Add a new file named NuGet.Config into the solution directory, and update the contents to this:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="local-packages" value=".\\CarLibrary\\Publish" />
  </packageSources>
</configuration>
```

Remove the project references from the CSharpCarClient and the VisualBasicCarClient project, and then add package references like this (from the solution directory):

```
dotnet add CSharpCarClient package CarLibrary
dotnet add VisualBasicCarClient package CarLibrary
```

How .net Core locates assemblies (minus GAC now)

How .NET Core Locates Assemblies

So far in this book, all of the assemblies that you have built were directly related (except for the NuGet example you just completed). You either added a project reference or a direct reference between projects. In these cases (as well as the NuGet example), the dependent assembly was copied directly into the target directory of the client application. Locating the dependent assembly isn't an issue, since they reside on the disk right next to the application that needs them.

But what about the .NET Core framework? How are those located? Previous versions of .NET installed the framework files into the Global Assembly Cache (GAC), and all .NET applications knew how to locate the framework files.

However, the GAC prevents the side-by-side capabilities in .NET Core, so there isn't a single repository of runtime and framework files. Instead, the files that make up the framework are installed together in "C:\\Program Files\\dotnet" (on Windows), separated by version. Based on the version of the application (as specified in the csproj file), the necessary runtime and framework files are loaded for an application from the specified version's directory.

Specifically, when a version of the runtime is started, the runtime host provides a set of *probing paths* that it will use to find an application's dependencies. There are five probing properties (each of them optional) as listed in Table 16-1.

Table 16-1. Application Probing Properties

Option	Meaning in Life
TRUSTED_PLATFORM_ASSEMBLIES	List of platform and application assembly file paths
PLATFORM_RESOURCE_ROOTS	List of directory paths to search for satellite resource assemblies
NATIVE_DLL_SEARCH_DIRECTORIES	List of directory paths to search for unmanaged (native) libraries
APP_PATHS	List of directory paths to search for managed assemblies
APP_NI_PATHS	List of directory paths to search for native images of managed assemblies

System.Configuration namespace

- Can read own configurations on the form:
 - Key – Value
- Other parts in a config file
- Example:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<startup>
<supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
</startup>
<!-- Custom App settings -->
<appSettings>
<add key="TextColor" value="Green" />
<add key="RepeatCount" value="8" />
</appSettings>
</configuration>
```

```
...
AppSettingsReader ar = new AppSettingsReader();
int numbOfTimes = (int)ar.GetValue("RepeatCount", typeof(int));
string textColor = (string)ar.GetValue("TextColor", typeof(string));
...
```

Example 2:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="IpAdresse" value="127.0.0.1"/>
  </appSettings>
</configuration>
```

App.config

```
...
using System.Configuration;
...
AppSettingsReader asr = new AppSettingsReader();
string ipadr = (string)asr.GetValue("IpAdresse", typeof(string));
...
```

Configuration schema documentation

- Config files can have many instructions
- Have notice on the XML structure
- See .NET help

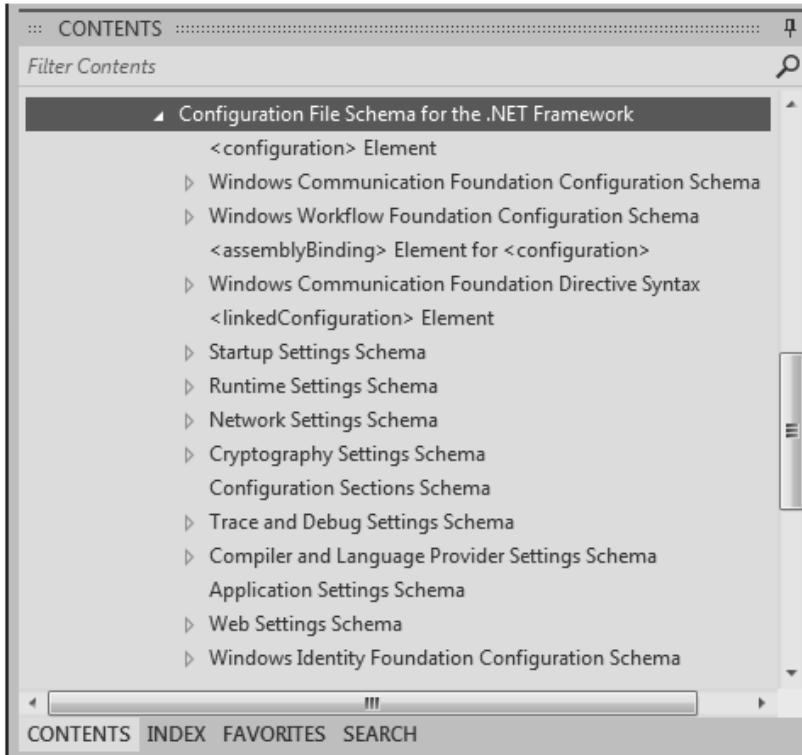


Figure 14-25. XML configuration files are fully documented in the .NET help system

About AddModule (source: Microsoft)

Visual C# Language Concepts

/addmodule (Import Metadata)

/addmodule:*file*[;*file2*]

where:

file, *file2*

An output file that contains metadata. The file cannot contain an assembly manifest. To import more than one file, separate file names with either a comma or a semicolon.

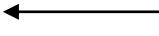
Remarks

All modules added with **/addmodule** must be in the same directory as the output file at run time. That is, you can specify a module in any directory at compile time but the module must be in the application directory at run time. If the module is not in the application directory at run time, you will get a **System.TypeLoadException**.

file cannot contain an assembly. For example, if the output file was created with **/target:module**, its metadata can be imported with **/addmodule**.

If the output file was created with a **/target** option other than **/target:module**, its metadata cannot be imported with **/addmodule** but can be imported with **/reference**.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio; a project cannot reference a module. 

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

Example

Compile source file `input.cs` and add metadata from `metad1.netmodule` and `metad2.netmodule` to produce `out.exe`:

```
csc /addmodule:metad1.netmodule;metad2.netmodule /out:out.exe input.cs
```

Type reflection and late binding

- The basic unit is assemblies
 - information can be read using programming
 - METADATA

Type Metadata

- Type metadata is a description of types
 - For example. classes, interfaces
 - information we can read on run-time
 - Type Metadata is also readable using ildasm.exe
-
- Every type is defined by a TypeDef token
 - If the Type using a reference it also have a TypeRef token
 - Extends token shows inheritance
 - Variables can be seen (Field)
 - Methods can be seen
 - return values
 - arguments
 - Different flags (for instance public)
 - Properties with set and get

TypeRef

- See example with System.Enum typeref block (page 565)

Assembly documentation

- MANIFEST
 - name
 - Public key
 - version number
 - Flags

External reference to other assemblies

- AssemblyRef in for instance System.Windows.Forms
- Describes external assembly
 - name
 - Public Key
 - version number
 - Hash value
 - Flags

String constants

- Is documented in "User String Tokens"

Reflection

Reflection is to find information about an assembly on run-time

- It is possible to find info about methods, variables and other info on run-time
 - for example: Method parameters
 - Which interfaces are used (implemented)

Use:
System.Type class
System.Reflection namespace

- Reflection namespace

Table 15-1. A Sampling of Members of the System.Reflection Namespace

Type	Meaning in Life
Assembly	This abstract class contains a number of static methods that allow you to load, investigate, and manipulate an assembly.
AssemblyName	This class allows you to discover numerous details behind an assembly's identity (version information, culture information, and so forth).
EventInfo	This abstract class holds information for a given event.
FieldInfo	This abstract class holds information for a given field.
MemberInfo	This is the abstract base class that defines common behaviors for the EventInfo, FieldInfo, MethodInfo, and PropertyInfo types.
MethodInfo	This abstract class contains information for a given method.
Module	This abstract class allows you to access a given module within a multifile assembly.
ParameterInfo	This class holds information for a given parameter.
PropertyInfo	This abstract class holds information for a given property.

The System.Type class

- The class gives much information
 - See selected members:

Table 15-2. Select Members of System.Type

Type	Meaning in Life
IsAbstract	These properties (among others) allow you to discover a number of basic traits about the Type you are referring to (e.g., if it is an abstract entity, an array, a nested class, and so forth).
IsArray	
IsClass	
IsCOMObject	
IsEnum	
IsGenericTypeDefinition	
IsGenericParameter	
IsInterface	
IsPrimitive	
IsNestedPrivate	
IsNestedPublic	
IsSealed	
IsValueType	
GetConstructors()	These methods (among others) allow you to obtain an array representing the items (interface, method, property, etc.) you are interested in. Each method returns a related array (e.g., GetFields() returns a FieldInfo array, GetMethods() returns a MethodInfo array, etc.). Be aware that each of these methods has a singular form (e.g., GetMethod(), GetProperty(), etc.) that allows you to retrieve a specific item by name, rather than an array of all related items.
GetEvents()	
GetFields()	
GetInterfaces()	
GetMembers()	
GetMethods()	
GetNestedTypes()	
GetProperties()	
FindMembers()	This method returns a MemberInfo array based on search criteria.
GetType()	This static method returns a Type instance given a string name.
InvokeMember()	This method allows “late binding” for a given item. You’ll learn about late binding later in this chapter.

Example:

```
using System;

namespace ENTYPE
{
    public class EnType
    {
        private string sLocal;
        public string sGlobal; // Only with the purpose
                             // of showing fields

        public EnType()
        {
            sGlobal ="UHA";
        }
    }
}
```

```
public EnType(string s)
{
    sLocal = s;
}

public void EnMetode(string s)
{
    sLocal = s;
}

}
```

L11_1.cs

```
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\csc.exe /t:library
/out:EnType.dll C:\undervisning\E2004\PCDN\lektioner\lektion11\l11_1.cs
```

- EnType.dll created

```
using System;
using System.Reflection;

public class Tester
{

    public static void Main()
    {
        Assembly a = null;

        try{

            a = Assembly.Load("EnType");
            Type Check = a.GetType("ENTYPE.EnType");

            MethodInfo[] mi = Check.GetMethods();
        }
    }
}
```

```
        for(int i = 0 ; i <mi.Length;i++)
            Console.WriteLine(mi[i].ToString());

            FieldInfo[] fi = Check.GetFields();

            for(int i = 0 ; i <fi.Length;i++)
                Console.WriteLine(fi[i].ToString());


        }catch(Exception e)
        {

            Console.WriteLine(e.Message);

        }

        Console.ReadLine();

    }

}
```

Output:

```
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void EnMetode(System.String)
System.Type GetType()
```

System.String sGlobal (uha!!)

- In the same way: Read constructors
 - and other parts

To create a Type

- System.Object has a method
 - GetType
 - returns an instance of a Type
 - Example

```
Auto skoda = new Auto();
Type t = skoda.GetType();
```

//Or:

```
Type t = typeof(Auto);
```

- Demands **compile-time knowledge** about Auto

Type.GetType()

- Type is an abstract class
 - cannot create object of Type
- Has a static method GetType
- Does **not demand compile-time knowledge** (points at an external assembly)

Type.GetType

- Comma separated arguments
- and concatenated arguments (nested types)

Other interesting methods and types in Type

method GetInterfaces()

- Which interfaces are supported
 - returns Type[]
- The BaseType
 - returns base class name
- IsAbstract
- IsSealed
- IsClass

The System.Reflection namespace

- Interesting classes
 - Assembly
 - AssemblyName
 - EventInfo
 - FieldInfo
 - MemberInfo
 - Module
 - ParameterInfo
 - PropertyInfo

AssemblyName example:

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

namespace SMall
{
    public class SmallClass
    {
        public double Plus(double a, double b)
        {
            return a + b;
        }
        public double Minus(double a, double b)
        {
            return a - b;
        }
    }
}
```

SMall.dll

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

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

```

{
    Assembly a = null;

    AssemblyName aName = new AssemblyName();
    try{
        Console.WriteLine("Enter DLL Name");
        string sName = Console.ReadLine();
        aName.Name = sName;
        a = Assembly.Load(aName); // load DLL
        Type[] types = a.GetTypes(); // Types in DLL ?
        Type type = a.GetType(types[0].FullName);

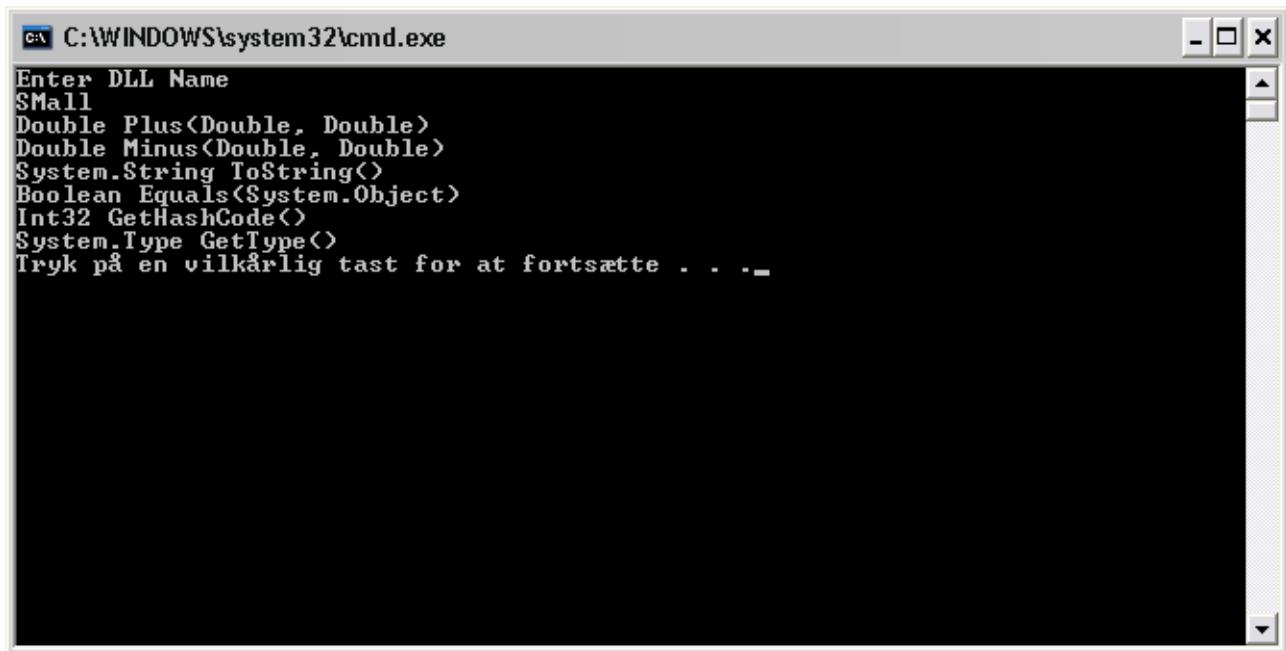
        MethodInfo[] mi = type.GetMethods();

        for (int i = 0; i < mi.Length; i++)
            Console.WriteLine(mi[i].ToString());

    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
}

...

```



- FullName = namespace + class name

Use interfaces

- Use interfaces to define (future) methods

Reflection on methods parameters and return values

Example:

- Reusing the same DLL

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

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

            Assembly a = null;
            AssemblyName aName = new AssemblyName();
            try{
                Console.WriteLine("Enter DLL Name");
                string sName = Console.ReadLine();
                aName.Name = sName;
                a = Assembly.Load(aName);           // load DLL
                Type[] types = a.GetTypes();        // Types in DLL ?
                Type type = a.GetType(types[0].FullName);
                MethodInfo[] mi = type.GetMethods();

                for (int i = 0; i < mi.Length; i++)
                {
                    Console.WriteLine(mi[i].ToString());
                    ParameterInfo[] pi = mi[i].GetParameters();
                    for (int y = 0; y < pi.Length; y++)
                        Console.WriteLine("PARAMTYPE: {0}, PARAMNAME: {1}",
                            pi[y].ParameterType, pi[y].Name);

                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
Enter DLL Name
Small
Double Plus<Double, Double>
PARAMTYPE: System.Double, PARAMNAME: a
PARAMTYPE: System.Double, PARAMNAME: b
Double Minus<Double, Double>
PARAMTYPE: System.Double, PARAMNAME: a
PARAMTYPE: System.Double, PARAMNAME: b
System.String ToString()
Boolean Equals<System.Object>
PARAMTYPE: System.Object, PARAMNAME: obj
Int32 GetHashCode()
System.Type GetType()
```

LINQ can be used to fetch a MethodInfo objects (or other) information

```
static void ListMethods(Type t)
{
Console.WriteLine("***** Methods *****");
var methodNames = from n in t.GetMethods() select n;
foreach (var name in methodNames)
Console.WriteLine("->{0}", name);
Console.WriteLine();
}
```

Dynamic invocation – (late binding)

- **Example above uses late binding**
- A type and members can be created on run-time
 - Errors can happen
 - But gives great flexibility

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

namespace SMall
{

    public class SmallClass
    {
        public double Plus(double a, double b)
        {
            return a + b;
        }
        public double Minus(double a, double b)
        {
            return a - b;
        }
    }
}
```

```

    }
}
```

Small.dll

```

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

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

            Assembly a = null;
            AssemblyName aName = new AssemblyName();
            try{
                Console.WriteLine("Enter DLL Name");
                string sName = Console.ReadLine();
                aName.Name = sName;
                a = Assembly.Load(aName); // load DLL
                Type[] types = a.GetTypes(); // Types in DLL ?
                for (int d = 0; d < types.Length; d++)
                {
                    Type type = a.GetType(types[d].FullName);
                    MethodInfo[] mi = type.GetMethods();

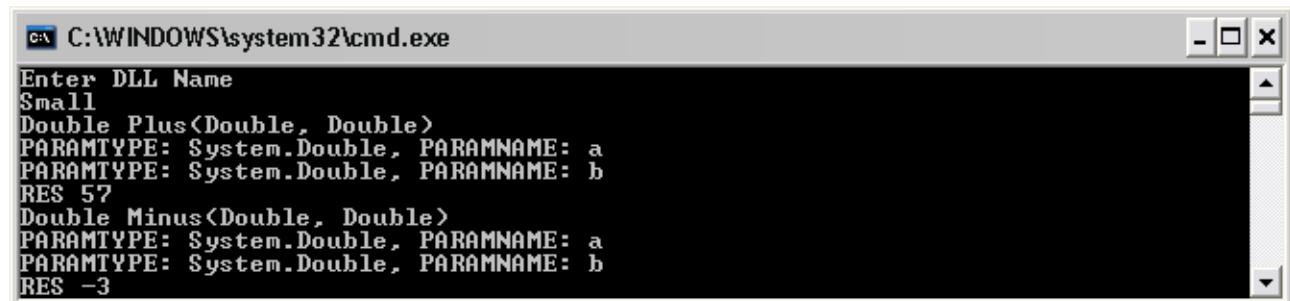
                    for (int i = 0; i < mi.Length; i++)
                    {
                        Console.WriteLine(mi[i].ToString());
                        ParameterInfo[] pi = mi[i].GetParameters();
                        for (int y = 0; y < pi.Length; y++)
                        {
                            Console.WriteLine("PARAMTYPE: {0}, PARAMNAME: {1}",
                                pi[y].ParameterType, pi[y].Name);
                        }
                    }

                    object[] inParams = new object[2]; // Maybe decided via contract
                    inParams[0] = 27.0;
                    inParams[1] = 30.0;
                    // CreateInstance
                    object calcObj = Activator.CreateInstance(types[d]);
                    // Get methods (Plus and Minus) (one at a time )
                    MethodInfo plus = types[d].GetMethod(mi[i].Name);
                    // Invoke
                    object ores = plus.Invoke(calcObj, inParams);
                    double res = (double)ores;
                    Console.WriteLine("RES {0}", res);
                }
            }
        }
    }
}
```

```
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
}
```

Notice the Activator class CreateInstance method

Output:



The screenshot shows a Windows Command Prompt window titled 'cmd' with the path 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text:

```
C:\WINDOWS\system32\cmd.exe
Enter DLL Name
Small
Double Plus<Double, Double>
PARAMTYPE: System.Double, PARAMNAME: a
PARAMTYPE: System.Double, PARAMNAME: b
RES 57
Double Minus<Double, Double>
PARAMTYPE: System.Double, PARAMNAME: a
PARAMTYPE: System.Double, PARAMNAME: b
RES -3
```

Reflection on shared assemblies

- A shared assembly cannot be loaded as simple as shown above
- MUST HAVE A PublicKeyToken

More about Late Binding

- Without any hard-code be able to activate and use assemblies
- Advantage: Early Binding : Errors found on compile time
- Advantage: Late Binding : flexible
- Activator.CreateInstance creates an object in memory

Arguments to methods

```

...
object[] inParams = new object[2]; // Maybe decided via contract
    inParams[0] = 27.0;
    inParams[1] = 30.0;
    // CreateInstance
    object calcObj = Activator.CreateInstance(types[d]);
    // Get methods (Plus and Minus) (one at a time )
    MethodInfo plus = types[d].GetMethod(mi[i].Name);
    // Invoke
    object ores = plus.Invoke(calcObj, inParams);
...

```

- NO ARGUMENTS?: use null; `object ores = plus.Invoke(calcObj, null);`

Attribute Programming

- Attributes are IDL keywords in array [] parenthesis
- Expand the metadata
- .NET attributes are classes
- System.Attribute is used

Table 15-3. A Tiny Sampling of Predefined Attributes

Attribute	Meaning in Life
<code>[CLSCompliant]</code>	Enforces the annotated item to conform to the rules of the Common Language Specification (CLS). Recall that CLS-compliant types are guaranteed to be used seamlessly across all .NET programming languages.
<code>[DllImport]</code>	Allows .NET code to make calls to any unmanaged C- or C++-based code library, including the API of the underlying operating system. Do note that <code>[DllImport]</code> is not used when communicating with COM-based software.

[Obsolete]	Marks a deprecated type or member. If other programmers attempt to use such an item, they will receive a compiler warning describing the error of their ways.
[Serializable]	Marks a class or structure as being “serializable,” meaning it is able to persist its current state into a stream.
[NonSerialized]	Specifies that a given field in a class or structure should not be persisted during the serialization process.
[ServiceContract]	Marks a method as a contract implemented by a WCF service.

AssemblyInfo.cs

- Easy to place attributes

Table 15-4. Select Assembly-Level Attributes

Attribute	Meaning in Life
[AssemblyCompany]	Holds basic company information
[AssemblyCopyright]	Holds any copyright information for the product or assembly
[AssemblyCulture]	Provides information on what cultures or languages the assembly supports
[AssemblyDescription]	Holds a friendly description of the product or modules that make up the assembly
[AssemblyKeyFile]	Specifies the name of the file containing the key pair used to sign the assembly (i.e., establish a strong name)
[AssemblyProduct]	Provides product information
[AssemblyTrademark]	Provides trademark information
[AssemblyVersion]	Specifies the assembly’s version information, in the format <major.minor.build.revision>

Several ways to set attributes

```
...
[Serializable, Obsolete("Use another vehicle!")]
public class HorseAndBuggy
{
    ...
}

[Serializable]
[Obsolete("Use another vehicle!")]
```

```
public class HorseAndBuggy
{
// ...
}
```

Homemade attribute

```
...
public sealed class ObsoleteAttribute : Attribute
{
    public ObsoleteAttribute(string message, bool error);
    public ObsoleteAttribute(string message);
    public ObsoleteAttribute();
    public bool IsError { get; }
    public string Message { get; }
}
Understand
...
```

Attributes on run-time

Example: Static binding

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

namespace SSmall
{
    [ClassDescription("A VERY VERY small calculator")]
    public class SmallClass
    {
        public double Plus(double a, double b)
        {
            return a + b;
        }
        public double Minus(double a, double b)
        {
            return a - b;
        }
    }

    public sealed class ClassDescription:Attribute
    {
        public string Desc {get; set;}

        public ClassDescription(String message)
        {
            Desc = message;
        }
    }
}
```

```

    public ClassDescription()
    {
    }
}

}

```

Small.dll

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Reflection;
using SMall;

namespace UsingSmall
{
    class Program
    {
        static void Main(string[] args)
        {
            Type attrib = typeof(SmallClass);
            object[] asmAttribs = attrib.GetCustomAttributes(false);

            foreach (ClassDescription cd in asmAttribs)
            {
                Console.WriteLine(cd.Desc);

            }
        }
    }
}

```



Example: Dynamic Binding

- Same DLL as above!

```

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

```

```
namespace UsingSmall
{
    class Program
    {
        static void Main(string[] args)
        {
            Assembly a = null;
            AssemblyName aName = new AssemblyName();
            try
            {
                Console.WriteLine("Enter DLL Name");
                string sName = Console.ReadLine();
                aName.Name = sName;
                a = Assembly.Load(aName); // load DLL

                Type[] types = a.GetTypes();
                String s = "SMall.ClassDescription";
                Type Info = a.GetType(s);
                PropertyInfo prop = Info.GetProperty("Desc");
                object[] oprop = types[0].GetCustomAttributes(Info, false);

                Console.WriteLine("Property Info {0}", prop.GetValue(oprop[0], null));
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Output again:



Attribute behaviors and types

Attribute behaviors can applied to these types:

- All—Any application element
- Assembly—Attribute can be applied to an assembly
- Class—Attribute can be applied to a class
- Constructor—Attribute can be applied to a constructor
- Delegate—Attribute can be applied to a delegate
- Enum—Attribute can be applied to an enumeration
- Event—Attribute can be applied to an event
- Field—Attribute can be applied to a field
- Interface—Attribute can be applied to an interface
- Method—Attribute can be applied to a method
- Module—Attribute can be applied to a module
- Parameter—Attribute can be applied to a parameter
- Property—Attribute can be applied to a property
- ReturnValue—Attribute can be applied to a return value
- Struct—Attribute can be applied to a structure

What else can attribute programming be used for?

- "Third Party" software can be decided this way.
- IMPORTANT: All O.S. functions can be used (C/C++ DLL's)

To get functions in O.S using attribute programming

Example: (Beep)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.InteropServices;
using EnRegner;

namespace DllBruger
{
    class Program
    {
        [DllImport("kernel32.dll")]
        public static extern bool Beep(int frequency,int duration);
        static void Main(string[] args)
        {
            double res;
            EnRegner.EnRegner regner = new EnRegner();

            res = regner.Plus(23.7, 45.8);
            Console.WriteLine(res);

            res = regner.Minus(23.7, 45.8);
            Console.WriteLine(res);
            Beep(1200, 500);
        }
    }
}
```

Example 2 (FTP Client) :

```
using System;

//namespace
using System;
using System.IO;
using System.Runtime.InteropServices;

namespace ConsoleApplication1
{
    //HOW TO: use WinInet API to download a file from an
    //      FTP server

    class __FtpDownload
    {
        [DllImport("WININET", EntryPoint="InternetOpen",
                  SetLastError=true, CharSet=CharSet.Auto)]
        static extern IntPtr __InternetOpen(

```

```
        string IpszAgent,
        int dwAccessType,
        string IpszProxyName,
        string IpszProxyBypass,
        int dwFlags);

[DllImport("WININET", EntryPoint="InternetCloseHandle",
           SetLastError=true, CharSet=CharSet.Auto)]
static extern bool __InternetCloseHandle(IntPtr hInternet);

[DllImport("WININET", EntryPoint="InternetConnect",
           SetLastError=true, CharSet=CharSet.Auto)]
static extern IntPtr __InternetConnect(
    IntPtr hInternet,
    string IpszServerName,
    int nServerPort,
    string IpszUsername,
    string IpszPassword,
    int dwService,
    int dwFlags,
    int dwContext);

[DllImport("WININET", EntryPoint="FtpSetCurrentDirectory",
           SetLastError=true, CharSet=CharSet.Auto)]
static extern bool __FtpSetCurrentDirectory(
    IntPtr hConnect,
    string IpszDirectory);

[DllImport("WININET", EntryPoint="InternetAttemptConnect",
           SetLastError=true, CharSet=CharSet.Auto)]
static extern int __InternetAttemptConnect(int dwReserved);

[DllImport("WININET", EntryPoint="FtpGetFile",
           SetLastError=true, CharSet=CharSet.Auto)]
static extern bool __FtpGetFile(
    IntPtr hConnect,
    string IpszRemoteFile,
    string IpszNewFile,
    bool fFailIfExists,
    FileAttributes dwFlagsAndAttributes,
    int dwFlags,
    int dwContext);

const int ERROR_SUCCESS = 0;
const int INTERNET_OPEN_TYPE_DIRECT = 1;
const int INTERNET_SERVICE_FTP = 1;

static void Main()
{
```

```

        IntPtr inetHandle = IntPtr.Zero;
        IntPtr ftpconnectHandle = IntPtr.Zero;

try
{
    //check for inet connection
if (__InternetAttemptConnect(0) != ERROR_SUCCESS)
throw new InvalidOperationException("no connection to internet available");

    //connect to inet
inetHandle =
    __InternetOpen("billyboy FTP", INTERNET_OPEN_TYPE_DIRECT, null, null, 0);
if (inetHandle == IntPtr.Zero)
    throw new NullReferenceException("couldn't establish a connection to the internet");

    //connect to ftp.microsoft.com
ftpconnectHandle = __InternetConnect(
    inetHandle, "ftp.microsoft.com", 21, "anonymous",
    "myemail@yahoo.com", INTERNET_SERVICE_FTP,
    0, 0);
    if (ftpconnectHandle == IntPtr.Zero)
        throw new NullReferenceException("couldn't connect to microsoft.com");

    //set to desired directory on FTP server
    if (!
__FtpSetCurrentDirectory(ftpconnectHandle, "/deskapps"))
        throw new InvalidOperationException("couldn't set to desired directory");

    //download file from server
if (! __FtpGetFile(ftpconnectHandle, "readme.txt", "c:\\downloadedFile1.txt", false, 0, 0, 0))
    throw new IOException("couldn't download file");

    //success
Console.WriteLine("SUCCESS: file downloaded successfully");
}

catch(Exception ex)
{
    //print error message
Console.WriteLine("ERROR: " + ex.Message);
}

finally
{
    //close connection to ftp.microsoft.com
    if (ftpconnectHandle != IntPtr.Zero)

        __InternetCloseHandle(ftpconnectHandle);
    ftpconnectHandle = IntPtr.Zero;
}

```

```

        //close connection to inet
    if (inetHandle != IntPtr.Zero)

        __InternetCloseHandle(inetHandle);
    inetHandle = IntPtr.Zero;
}
}

/*
{
/// <summary>
/// Summary description for Class1.
/// </summary>
class Class1
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        //
        // TODO: Add code to start application here
        //
    }
}
*/

```

Example 3:

```

#define TRACE_ON ←
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.InteropServices; // for DllImport
using System.Diagnostics; // for Conditional

using EnRegner;

namespace DllBruger
{

    public class Sound

```

```
{  
    [DllImport("kernel32.dll")]  
    public static extern bool Beep(int frequency, int duration);  
  
    [Conditional("TRACE_ON")]  
    public static void Noise()  
    {  
        Beep(1200, 500);  
    }  
}  
  
class Program  
{  
  
    static void Main(string[] args)  
    {  
        double res;  
        EnRegner.EnRegner regner = new EnRegner.EnRegner();  
  
        res = regner.Plus(23.7, 45.8);  
        Console.WriteLine(res);  
  
        res = regner.Minus(23.7, 45.8);  
        Console.WriteLine(res);  
  
        Sound.Noise();  
    }  
}
```



Extendable applications

Example VS:

Putting Reflection, Late Binding, and Custom Attributes in Perspective

Even though you have seen numerous examples of these techniques in action, you may still be wondering when to make use of reflection, dynamic loading, late binding, and custom attributes in your programs. To be sure, these topics can seem a bit on the academic side of programming (which may or may not be a bad thing, depending on your point of view). To help map these topics to a real-world situation, you need a solid example. Assume for the moment that you are on a programming team that is building an application with the following requirement:

- The product must be extendable by the use of additional third-party tools.

What exactly is meant by *extendable*? Well, consider the Visual Studio IDE. When this application was developed, various “hooks” were inserted into the codebase to allow other software vendors to “snap” (or plug in) custom modules into the IDE. Obviously, the Visual Studio development team had no way to set references to external .NET assemblies it had not developed yet (thus, no early binding), so how exactly would an application provide the required hooks? Here is one possible way to solve this problem:

DYNAMIC LOAD

1. First, an extendable application must provide some input mechanism to allow the user to specify the module to plug in (such as a dialog box or command-line flag). This requires *dynamic loading*.

REFLECTION

2. Second, an extendable application must be able to determine whether the module supports the correct functionality (such as a set of required interfaces) to be plugged into the environment. This requires *reflection*.

LATE BINDING

3. Finally, an extendable application must obtain a reference to the required infrastructure (such as a set of interface types) and invoke the members to trigger the underlying functionality. This may require *late binding*.