# .NET to COM Interoperability

# Agenda

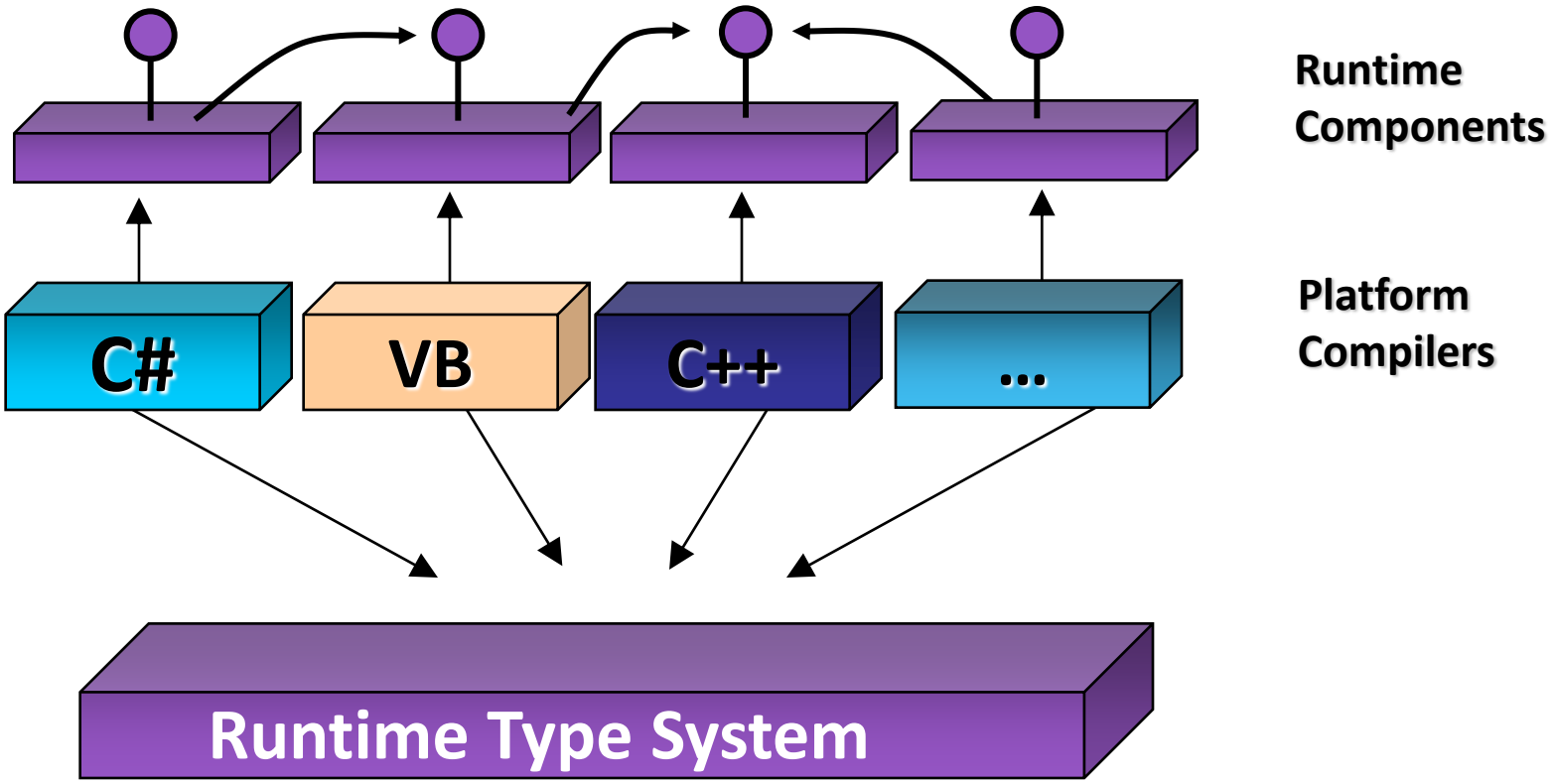
- The big picture
- Using COM types from .NET
- Using .NET types from COM

# Why Interop

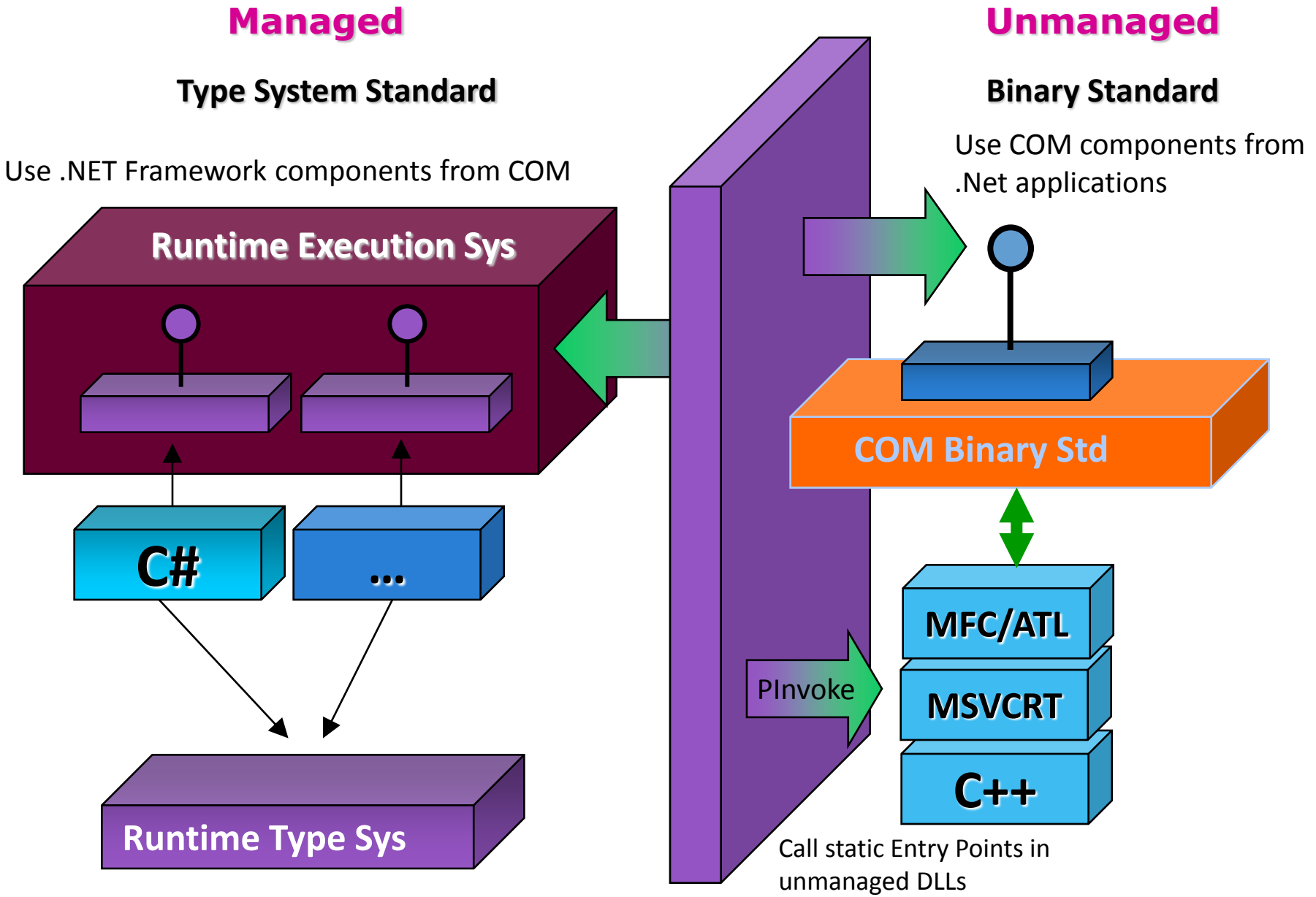
- Preserve/Utilize your investment
  - No need to start over
  - Continue to use exist code
- Incremental migration path
  - Migrate your application step by step
- Reality – some things never change
  - Need to interop with code that can't change

# Runtime Type System

.NET compilers use the Runtime Type System to produce *type compatible* components



# COM Interoperability Services



# Bridging Different Worlds

**Unmanaged Code**

- Binary standard
- Type Libraries
- DLL Hell
- Interface based
- HResults
- Guids

**Managed Code**

- Type Standard
- Metadata
- Assemblies
- Object/Interface based
- Exceptions
- Strong Names

# Model Consistency

- **Programming model remains consistent**
  - COM developers use COM model
  - .NET Framework developers use .NET Framework model
- **Model Transparency**
  - The CLR does the needed transformation

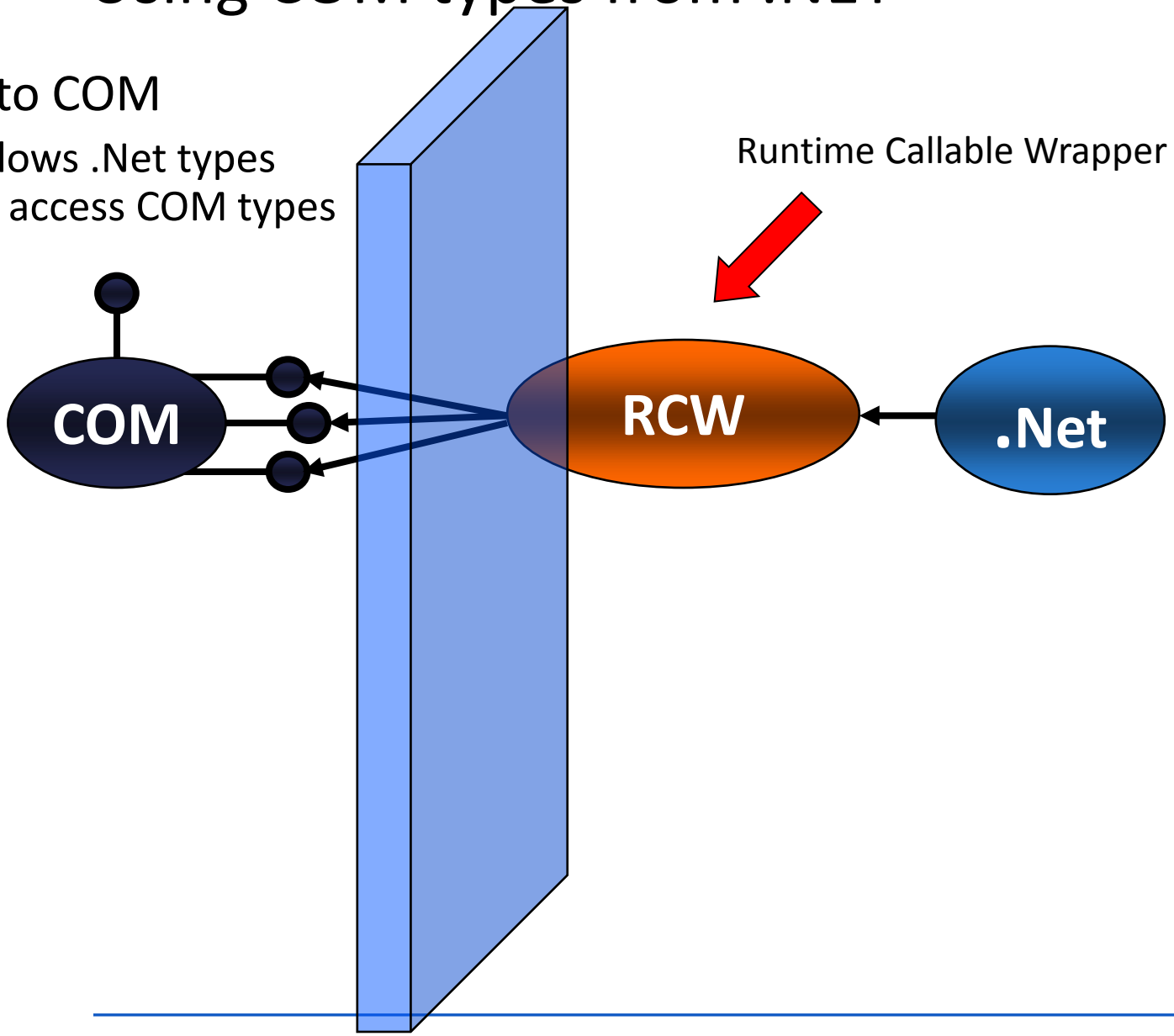
| .NET Framework | | COM Model |
|---|---|---|
| ❑ *new* operator | ⟺ | ☐ CoCreateInstance |
| ❑ Cast operator | ⟺ | ☐ QueryInterface |
| ❑ Memory mgmt | ⟺ | ☐ Reference Counting |
| ❑ Exceptions | ⟺ | ☐ Hresults |

# .NET to COM Interoperability

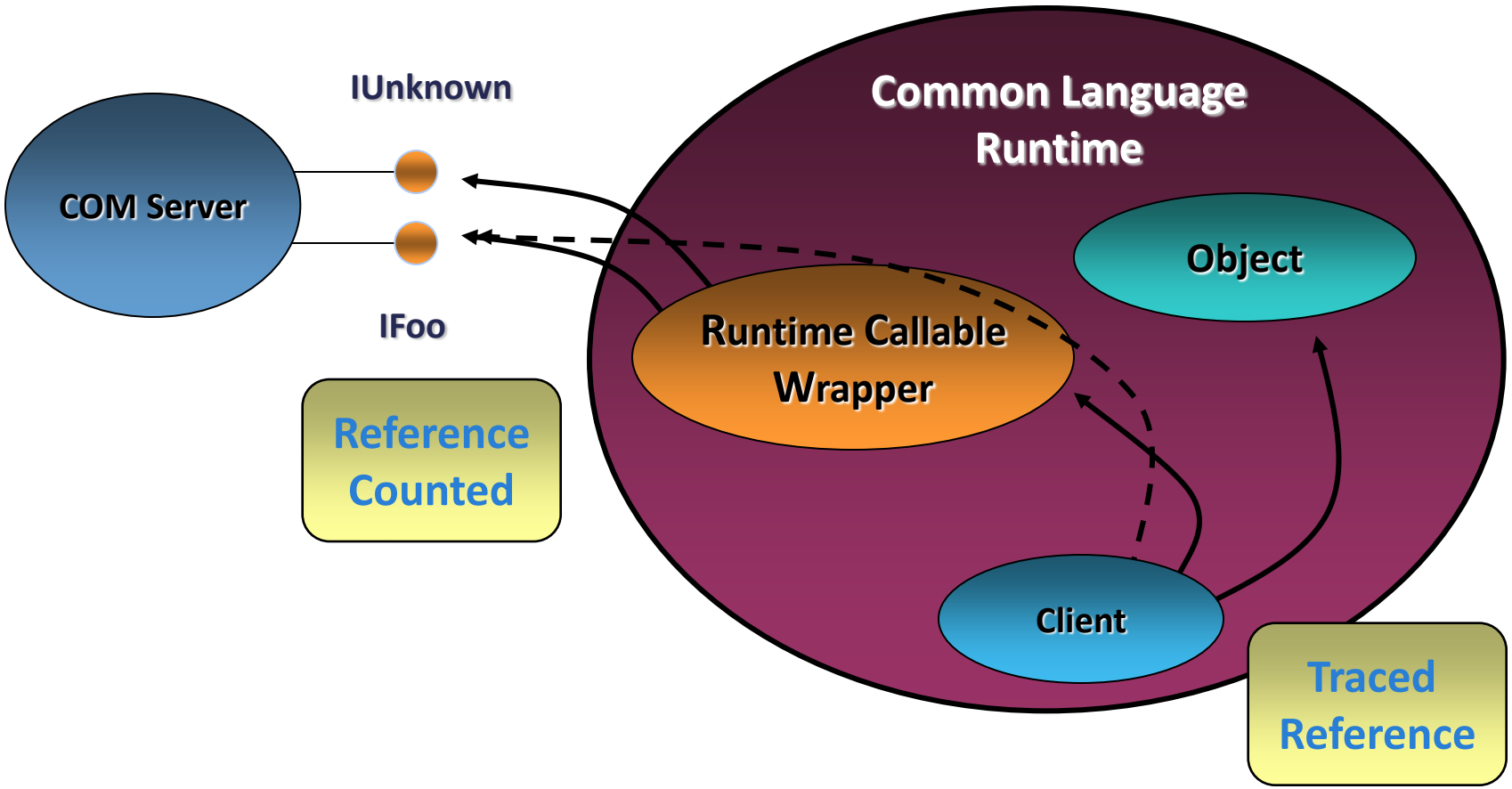
## Using COM types from .NET

# Using COM types from .NET

- .Net to COM
  - Allows .Net types to access COM types



# .Net To COM Interop

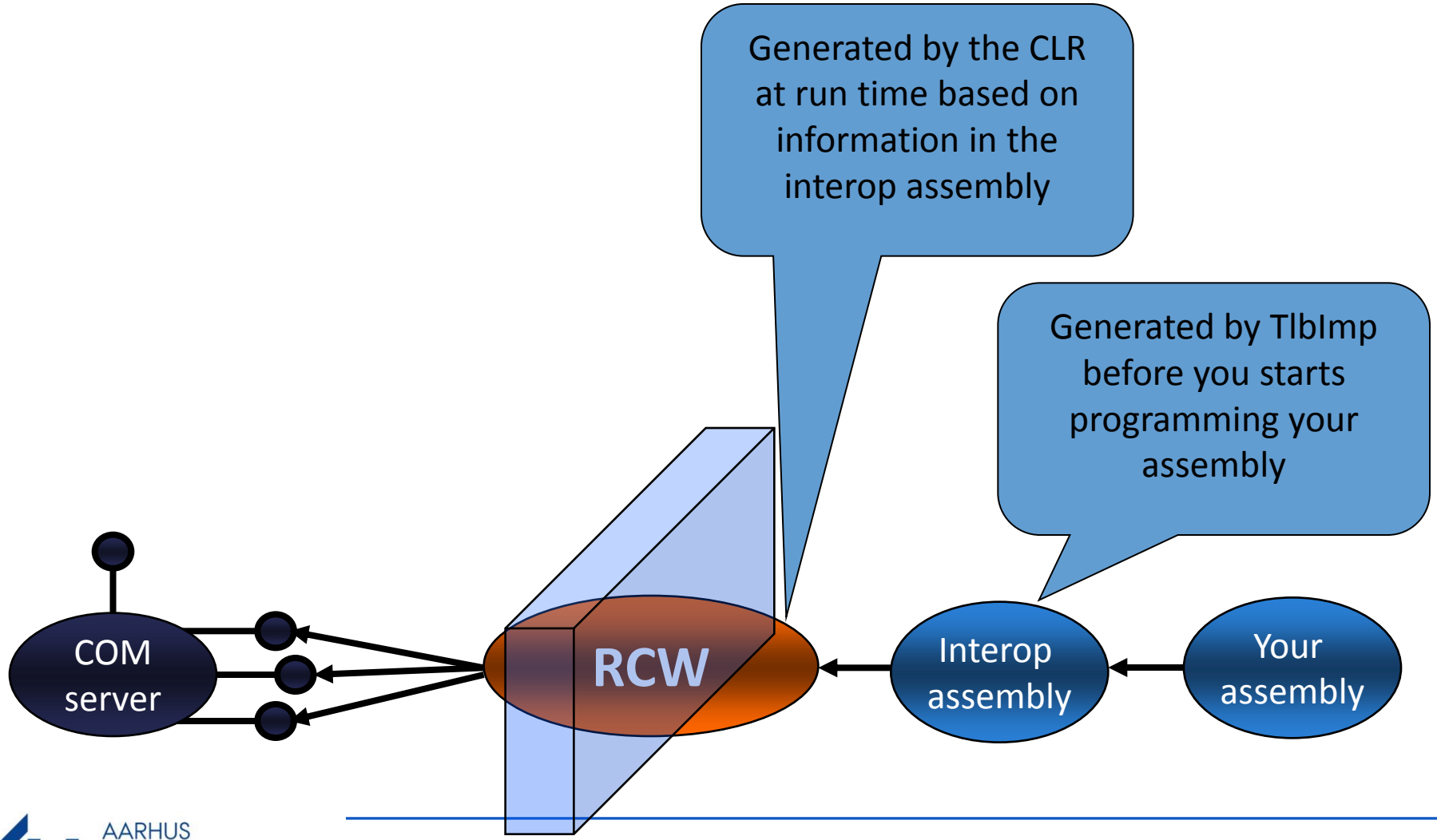


# Using COM Types From .NET

- Create an assembly containing type definitions for COM types (this assembly is called the Interop Assembly)
  - By adding references in VS
    - This is the easy way
  - By using type library importer (TlbImp.exe ← console tool!)
    - TLBIMP MyLib.tlb
    - Use TLBIMP when the default mappings isn't suitable.
  - By defining types manually
    - The hard way
- Reference the assembly from other applications
  - By adding references in VS
  - Or from the command prompt:
    - CSC /t:library /r:MyLib.dll MyCode.cs
- Use the types as managed types!
  - Create instances with new
  - Catch exceptions
  - Even extend them by use of inheritance

# The Interop Assembly

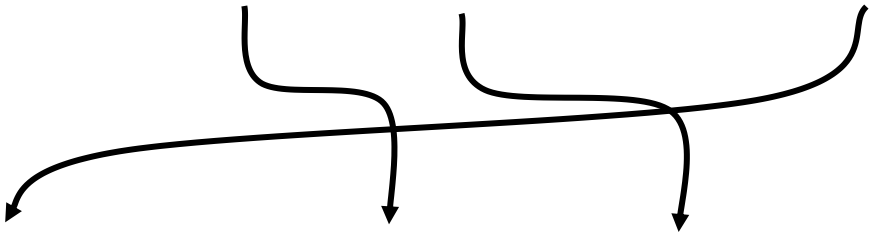
- The Interop Assembly contains metadata describing the COM types and methods that forward the call to COM.



# TlbImp Signature Translation

**COM Method Signature**

```
HRESULT FormatDate(BSTR s, DATE d, [out, retval] int *retval);
```



```
int FormatDate(String s, DateTime d);
```

**.NET Method Signature**

# Data Type Conversion

Performed by TlbImp

- DATE → System.DateTime
- BSTR → System.String
- SafeArray(int) → int[]
- OLECOLOR → System.Drawing.Color
- CURRENCY → System.Decimal
- VARIANT → System.Object

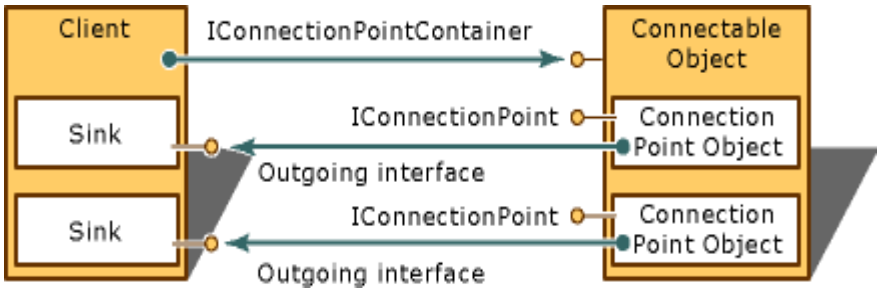
# Using COM Types From .NET

- Don’t think of types as COM types
  - No reference counting
  - No IDispatch
  - No Hresults
  - No Connection Points
  - No Guids
- Call directly through the class
  - Members of the default interface are added to the class during import
- Cast to specific interfaces as necessary
  - InvalidCastException if underlying QI fails
- Failure HResults are automatically mapped to exceptions
- Enumerations are automatically mapped to IEnumVariant

# Connection points

- A connectable object is one that supports outgoing interfaces.
- An outgoing interface allows the object to communicate with a client.
  - A connection point is like an event in .Net
- For each outgoing interface, the connectable object exposes a connection point.

- Each outgoing interface is implemented by a client on an object called a sink.
  - A sink serve the same function as an eventhandler in .Net



# Connection points in ATL

**ATL uses the following classes to support connection points:**

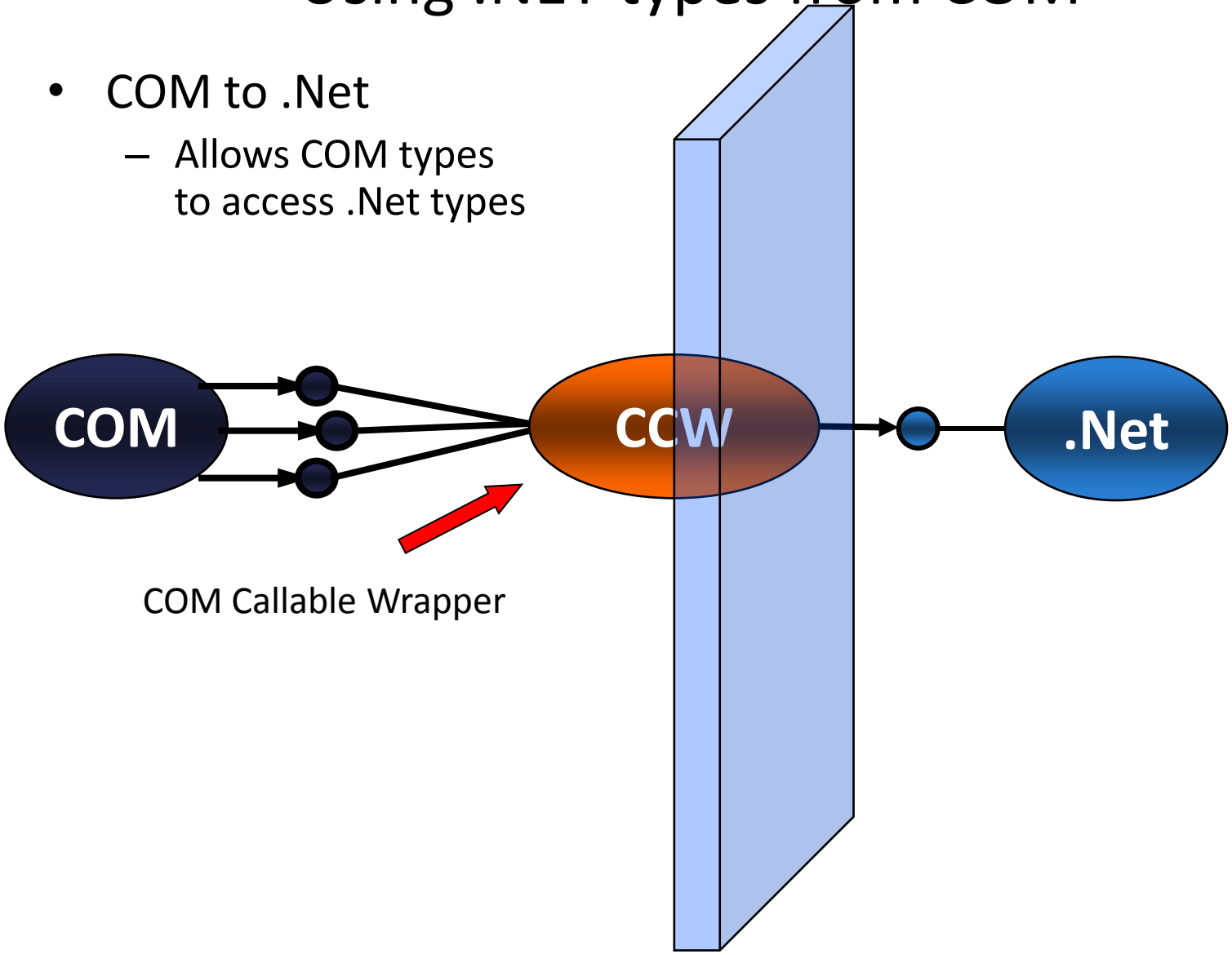
- IConnectionPointImpl implements a connection point. The IID of the outgoing interface it represents is passed as a template parameter.
- IConnectionPointContainerImpl implements the connection point container and manages the list of **IConnectionPointImpl** objects.
- IPropertyNotifySinkCP implements a connection point representing the IPropertyNotifySink interface.
- CComDynamicUnkArray manages an arbitrary number of connections between the connection point and its sinks.
- CComUnkArray manages a predefined number of connections as specified by the template parameter.
- CFirePropNotifyEvent notifies a client's sink that an object's property has changed or is about to change.
- IDispEventImpl provides support for connection points for an ATL COM object. These connection points are mapped with an event sink map, which is provided by your COM object.
- IDispEventSimpleImpl works in conjunction with the event sink map in your class to route events to the appropriate handler function.

# COM to .NET Interoperability

## Using .NET types from COM

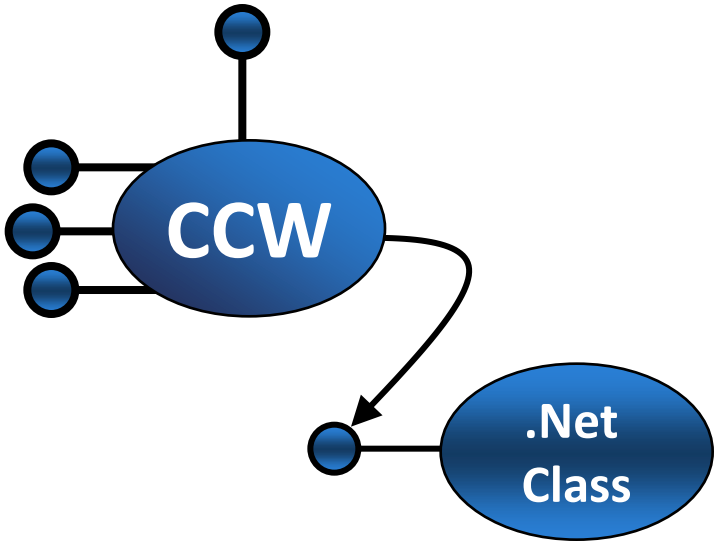
# Using .NET types from COM

- COM to .Net
  - Allows COM types to access .Net types



# COM Callable Wrapper (CCW)

- All .NET objects wrapped with a COM Callable Wrapper (CCW)
- The wrapper implements
  - IUnknown
  - IDispatch
  - ITypeInfo
  - IProvideClassInfo
  - IConnectionPoint, etc.



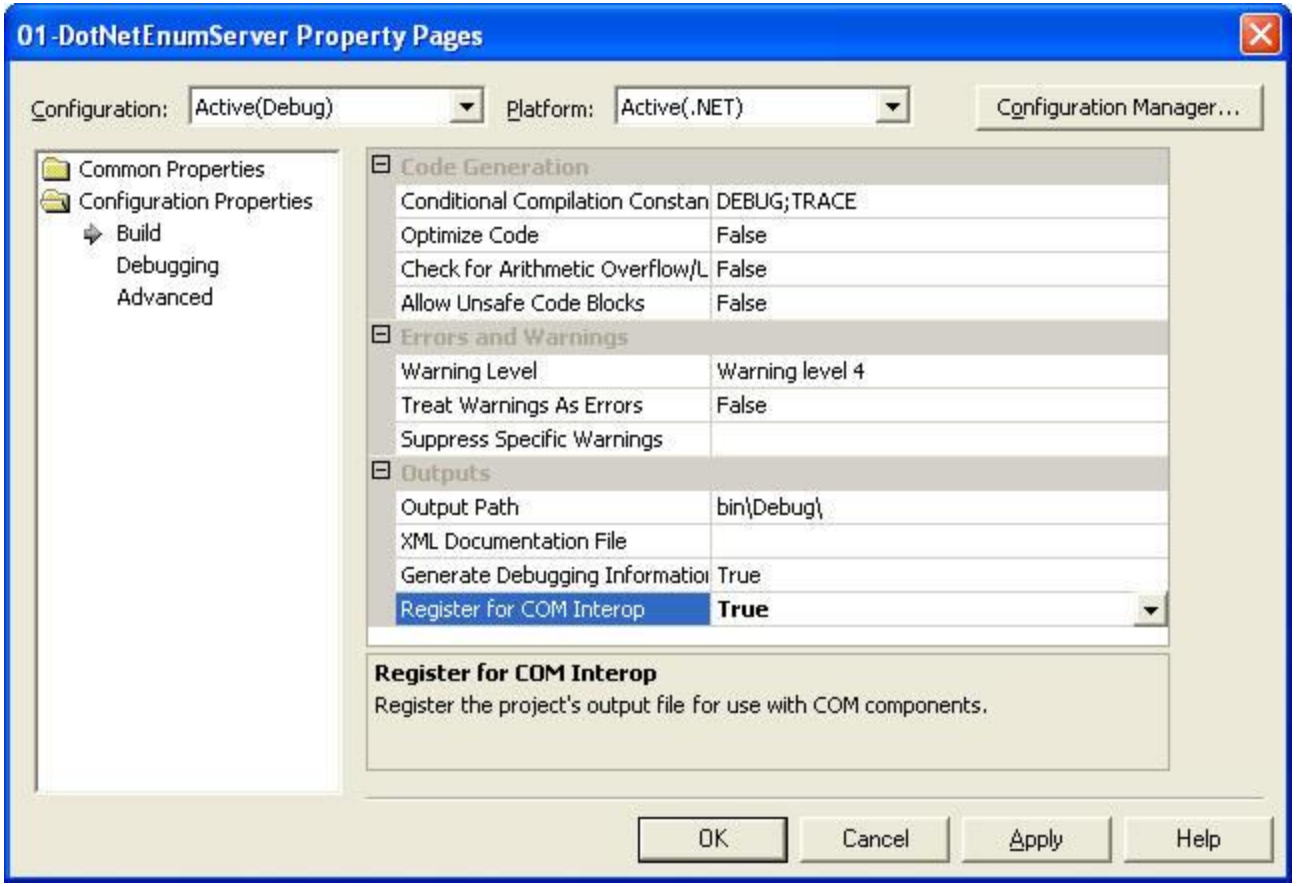

- Wrapper also manages identity, handles exceptions, signature conversion, etc

# Exporting .NET Types to COM

- Design for Interoperability
  - You can't call everything from COM.
- Create COM type library
  - TLBEXP MyLib.dll
    (no need for tlbexp if you use the /tlb option on the regasm utility)
- Install the assembly in global assembly cache
  - GACUTIL /i Mylib.dll
- Register the assembly in the system registry
  - RegAsm MyLib.dll
- **You can create a type library and register the assembly AND type library in on step, by use of RegAsm with the /tlb option!**
  - **RegAsm /tlb MyLib.dll**

# Exporting .NET Types to COM by VS

- When developing a .Net component mend to be callable from COM you can get Visual Studio to do the registration automatically.
  - Under Project Settings change Confuguration Properties - Register for COM Interop to true



# Using .NET Types from C++ through COM

- Reference the type libraries as necessary by use of #import
  - My experience with import of type libraries generated from .Net servers is that use of #import with the attribute no_namespace gives an “internal compiler error", so don’t use this option!
  - If you don’t call CoCreateInstance directly, but only through smart pointers you can omit the named_guids options.
  - If the server uses types from the .Nets base class library (they very often do that), then you also need to import the type library for mscorlib (and maybe others)

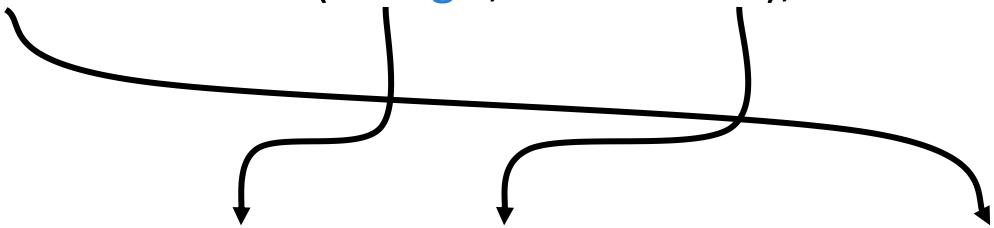
- Eg. to use the DotNetEnumServer you should write:

```
#import "C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\mscorlib.tlb" \
rename("ReportEvent", "ReflectionReportEvent")
#import "..\01-DotNetEnumServer\bin\Debug\DotNetEnumServer.tlb"
// import will place all data types in namespaces with names after the containing  type library
using namespace mscorlib;
using namespace DotNetEnumServer;
```

# TlbExp Signature Translation

## .NET Method Signature

```
int FormatDate(String s, DateTime d);
```



```
HRESULT FormatDate([in] BSTR s, [in] DATE d, [out, retval] int *retval);
```

## COM Method Signature

# Data Type Conversion

Performed by TLBEXP

| | | |
|---|---|---|
| • DATE | ⟸ | System.DateTime |
| • BSTR | ⟸ | System.String |
| • Safearray(int) | ⟸ | int [ ] |
| • OLECOLOR | ⟸ | System.Drawing.Color |
| • CURRENCY | ⟸ | System.Decimal |
| • VARIANT | ⟸ | System.Object |

# Helpful Information

- Not all managed types are accessible
  - Only public types and members are exposed
  - Static members are not accessible
  - Creatable classes must have default constructor
  - Expose functionality through interfaces
- Guid's generated automatically at export
  - Based on assembly and type name
  - Based on complete interface definition
- Assembly must be resolvable at runtime
  - Installed in application directory or
  - Installed in global assembly cache (GAC)

# Custom Attributes

Custom Attributes can be applied to types, methods, properties, fields or parameters to effect the COM type definitions produced by TlbExp.

```
using System.Runtime.InteropServices;

[Guid(…), InterfaceType(ComInterfaceType.IsIUnknown)]
Interface IFooBar {
   [DispId(64)] int Format( [MarshalAs(LPStr)] string s)
}
```

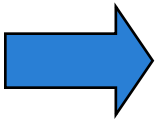
# Class Interfaces

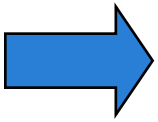
- A class interface can be generated at export
- Attribute can be applied to individual class or entire assembly to control class interface generation
- ClassinterfaceType.None
  - **Recommend approach**
  - No class interface produced
  - First implemented interface becomes the default
- ClassinterfaceType.AutoDispatch
  - Default setting
  - Class interface supported but no type information
  - Works well for script and late bound clients
- ClassinterfaceType.AutoDual
  - Not recommended
  - Automatic class interface produced

# AutoDual Class Interfaces

```
[ClassInterface(
ClassInterfaceType.AutoDual)]
public class A {
   void M1();
}
```

```
[ClassInterface(
ClassInterfaceType.AutoDual)]
public class B : A {
   void M2();
}
```





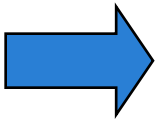
```
Coclass A {
     [default] interface _A;
}
Interface _A : IDispatch {
     …
     HRESULT M1();
}

Coclass B {
     [default] interface _B;
}
Interface _B : IDispatch {
     …
     HRESULT M1();
     HRESULT M2();
}
```

Use With Caution

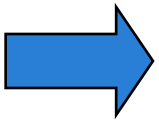
# AutoDispatch Class Interfaces

```
[ClassInterface(
ClassInterfaceType.AutoDisp]
public class A {
   void M1();
}
```



```
Coclass A {
     [default] IDispatch;
}
```

```
[ClassInterface(
ClassInterfaceType.AutoDisp]
public class B : A {
   void M2();
}
```



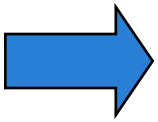
```
Coclass B {
     [default] IDispatch;
}
```

# No Class Interface

```
Interface IA {
   int M1();
   int M2();
}


[ClassInterface(
ClassInterfaceType.None)]
public class A : IA {
   int M1();
   int M2();
}
```

First inherited interface becomes the default interface

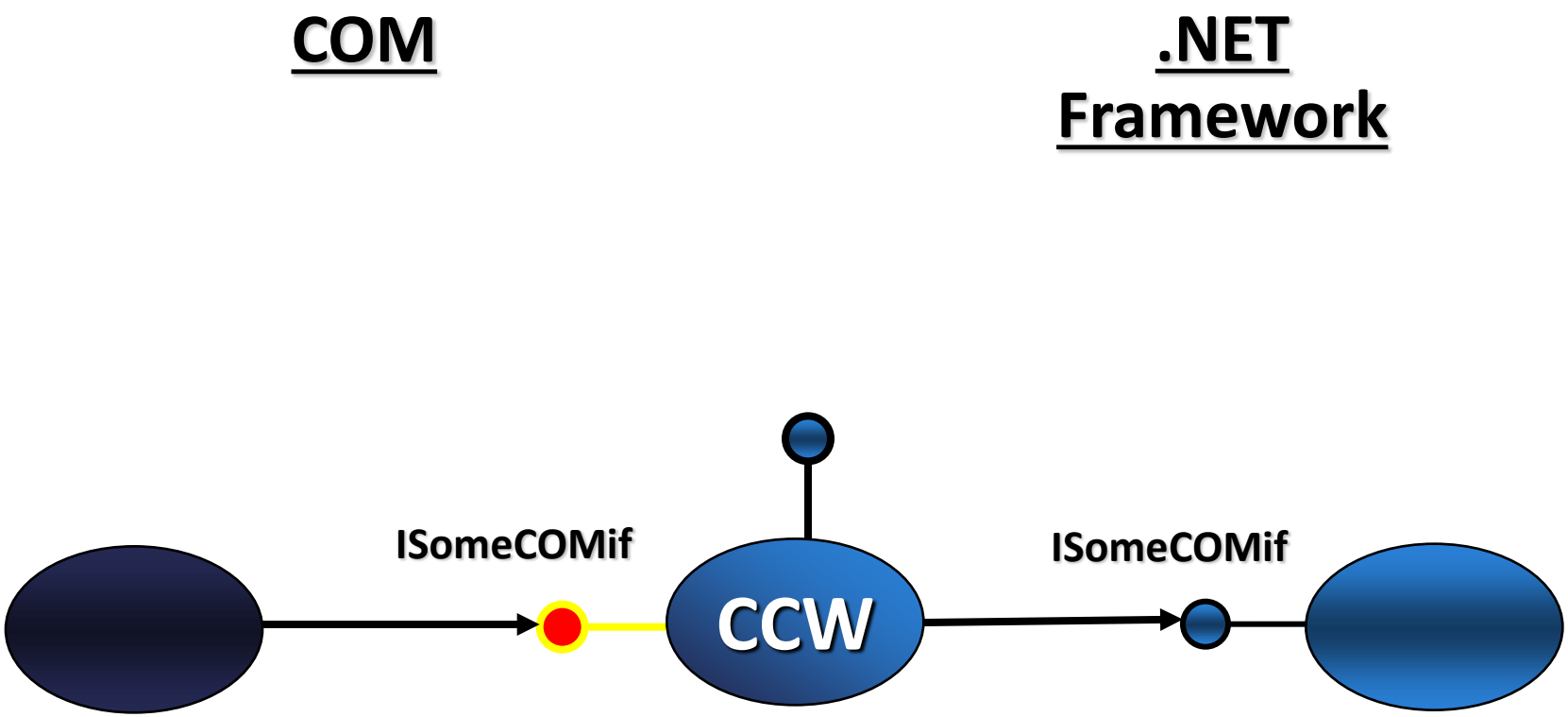


```
Coclass A {
     [default] interface IA;
}

Interface IA : IDispatch {
     HRESULT M1();
     HRESULT M2();
}
```

*Implementing functionality through interfaces has a major benefit for COM clients. .NET keeps interfaces consistent with previous versions when generating CCWs. This helps keep changes to your .NET server from breaking COM clients.*

# Building COM Compatible Types



# Building COM Compatible Types

- Provide compatible implementations of existing COM interfaces
- Interfaces must have compatible layout, IID, DispID, method signature
- Start by importing with TlbImp
- Implement necessary interfaces
- Export with TlbExp and register with RegAsm

# Enums

```
// Dotnetenumserver.cs
public enum CarMake : byte
  {
    BMW = 10, Dodge = 20,
    Saab = 30, VW = 40,
    Yugo = 0
  }

[ClassInterface(ClassInterfaceType.AutoDual)]
[Guid("F64F79EA-DF4C-48d3-97AF-534A7F197EDD")]
public class Car
  {
    public Car(){}
    private CarMake mCarMake = CarMake.BMW;
    public CarMake CarMake
        {
          get {return mCarMake;}
          set {mCarMake = value;}
        }
…
```

```
Debug>regasm /tlb dotnetenumserver.dll
Debug>gacutil /i dotnetEnumserver.dll
```

# Enums – the generated IDL