Component Programming with C# and .NET

- 1st Class Component Support
- Robust and Versionable
- Creating and using attributes
- API integration
  - DLL import
  - COM support
- Preserve Existing Investments

What defines a component?

- What defines a component?
  - Properties, methods, events
  - Design-time and runtime information
  - Integrated help and documentation
- C# has first class support
  - Not naming patterns, adapters, etc.
  - Not external files
- Easy to build and consume

Properties

- Properties are “smart fields”
  - Natural syntax, accessors, inlining

```
public class Button: Control
{
   private string caption;

   public string Caption {
      get {
         return caption;
      }
      set {
         caption = value;
         Repaint();
      }
   }
}
```

```
Button b = new Button();
b.Caption = "OK";
String s = b.Caption;
```

Indexers

- Indexers are “smart arrays”
  - Can be overloaded

```
public class ListBox: Control
{
   private string[] items;

   public string this[int index]{
      get {
         return items[index];
      }
      set {
         items[index] = value;
         Repaint();
      }
   }
}
```

```
ListBox listBox = new ListBox();
listBox[0] = "hello";
Console.WriteLine(listBox[0]);
```

Events

- Efficient, type-safe and customizable
  - Built on delegates

```
public class MyForm: Form {
  public MyForm()
  {
    Button okButton = new Button(...);

    okButton.Click += new EventHandler(OkButtonClick);
  }

  void OkButtonClick(…)
  {
    ShowMessage("You clicked OK");
  }
}
```

Design/Runtime Information

- Add information to types + methods?
  - Transaction required for a method?
  - Mark members for persistence?
  - Default event hookup?
- Traditional solutions
  - Lots of custom user code
  - Naming conventions between classes
  - External files (e.g. .IDL, .DEF)
- The C# solution - Attributes

Attributes

- Appear in square brackets
- Attached to code elements

```
[HelpUrl("http://SomeUrl/Docs/SomeClass")]
class SomeClass
{
   [WebMethod]
   void GetCustomers() { … }

   string Test([SomeAttr] string param1) {…}
}
```

Attribute Fundamentals

- Attributes are classes! Completely generic

```
class HelpUrl : System.Attribute {
    public HelpUrl(string url) { … }
    …
}
```

```
[HelpUrl("http://SomeUrl/APIDocs/SomeClass")
]
class SomeClass { … }
```

- Easy to attach to types and members

```
Type type = Type.GetType("SomeClass");
object[] attributes =
     type.GetCustomAttributes();
```

- Attributes can be queried at runtime

Attributes in .NET

- Web Services
- COM Interop
- Platform Invoke (DLL Interop)
- Transaction Contexts
- Permissions
- Custom XML Persistence

- **User-defined attributes to specify non-functional component properties** (RT / FT / Security / Config.)

XML Comments

- XML schema for comments
  - Method description
  - Parameter types, names and descriptions
  - Add your own tags - just XML
- Compiler and IDE support
  - Compiler creates XML file for all classes
  - IDE support for entry and display
- Used throughout .NET Framework

Using Attributes

```
[HelpUrl("http://SomeUrl/APIDocs/SomeClass")]
class SomeClass
{
    [Obsolete("Use SomeNewMethod instead")]
    public void SomeOldMethod()
    { … }

    public string Test( [SomeAttr()] string param1 )
    { … }
}
```

Using Attributes (contd.)

```
[HelpUrl("http://SomeUrl/MyClass")]
class MyClass  {}

[HelpUrl("http://SomeUrl/MyClass”, Tag=“ctor”)]
class MyClass  {}

[HelpUrl("http://SomeUrl/MyClass"),
 HelpUrl("http://SomeUrl/MyClass”, Tag=“ctor”)]
class MyClass  {}
```

Querying Attributes

```
Type type = typeof(MyClass);

foreach(object attr in type.GetCustomAttributes() )
{
    if ( attr is HelpUrlAttribute )
    {
        HelpUrlAttribute ha = (HelpUrlAttribute) attr;

        myBrowser.Navigate( ha.Url );
    }
}
```

Replication based on Attributes

- Specification of a component‘s non-functional properties at design time
- A tool may generate code to automatically create replicated objects
- Component behavior described by user-defined attributes

```
namespace CalculatorClass {
    using System; using proxy;

    [TolerateCrashFaults(4)]
    public class Calculator  {
            ...
            public double add
                (double x, double y) {
                    return x + y;
            }
    }
}
```

Behind the scenes...

```
public sealed class Calculator:CalculatorClass.Calculator {
        private CalculatorClass.Calculator[] _bc;
        public Calculator(): base() {
            _ErrorCount=0; int _Count=1;
            System.Attribute[] _arAtt =

System.Attribute.GetCustomAttributes(GetType());
            each(System.Attribute _attr in _arAtt) {
                if(_attr is TolerateCrashFaults)
                    _Count=((TolerateCrashFaults)_attr).Count;
            }
        /// creation of sufficiently many proxy objects
            _bc=new CalculatorClass.Calculator[_Count];
```

Reflection

Attribute-based programming

From the consumer‘s perspective

```
namespace CalculatorFront {
    using System;
    /// using CalculatorClass;
    using proxy;

    public class MainClass {
        public MainClass() {}
        public static int Main(string[] args) {
            Calculator calc = new Calculator();
                                ...
            result = calc.add(val1, val2);
```

Minimal code changes

Calling Into Existing DLLs

- .NET Framework contains attributes to enable calling into existing DLLs
- System.Runtime.InteropServices
  - DLL Name, Entry point, Parameter and Return value marshalling, etc.
- Use these to control calling into your existing DLLs
  - System functionality is built into Framework

Attributes to specify DLL Imports

```
[DllImport("gdi32.dll")]
public static extern
   int CreatePen(int style, int width, int color);
```

```
[DllImport("gdi32.dll",CharSet=CharSet.Auto)]
public static extern
   int GetObject(  int hObject,
                   int nSize,
                   [In, Out] ref LOGFONT lf);
```

COM Support

- .NET Framework provides great COM support
  - TLBIMP imports existing COM classes
  - TLBEXP exports .NET types
- Most users will have a seamless experience

Calling into a COM component

- Create .NET assembly from COM component via **tlbimp**
- Client apps may access the newly created assembly

```
using System;
using System.Runtime.InteropServices;

using CONVERTERLib;

class Convert {
        public static void Main(string [] args) {
        CFConvert conv = new CFConvert();
        ...
        fahrenheit = conv.CelsiusToFahrenheit( celsius );
```

COM Support

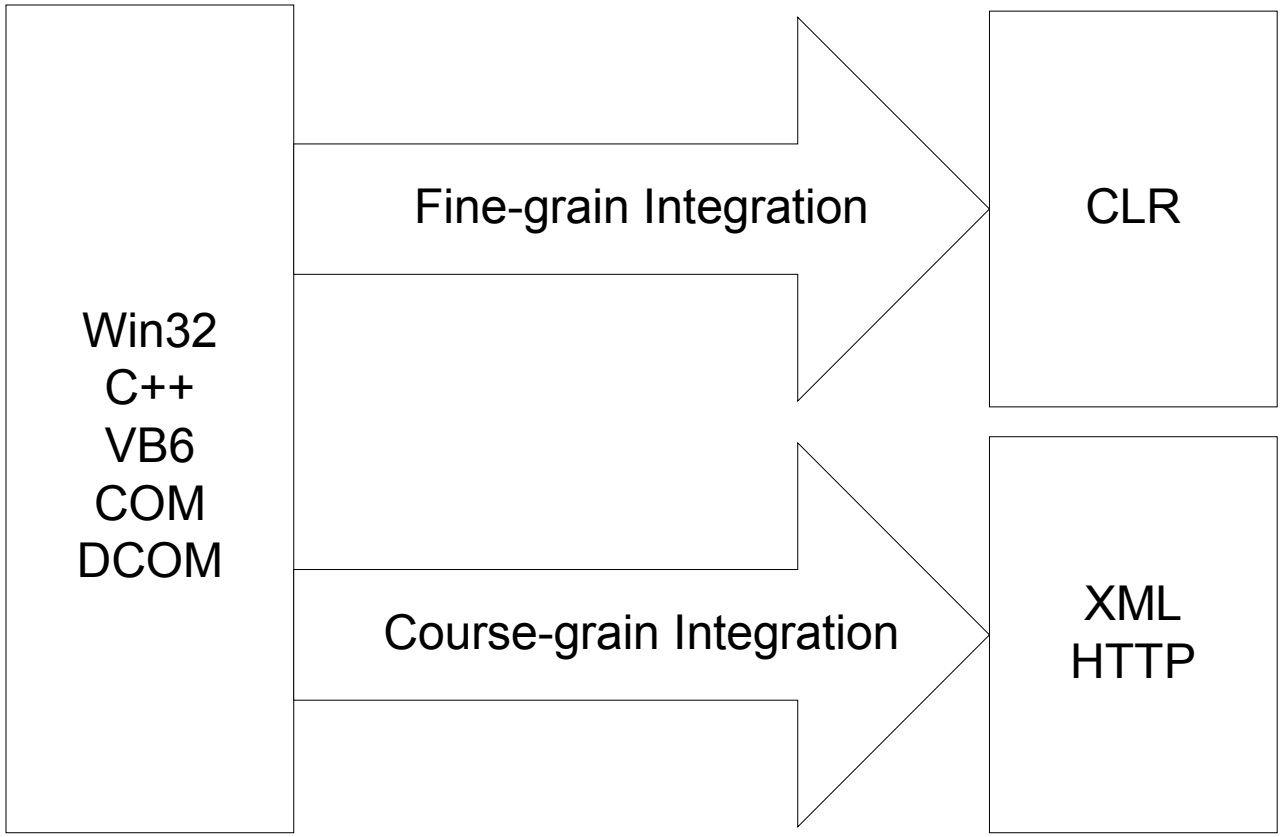
- Sometimes you need more control
  - Methods with complicated structures as arguments
  - Large TLB – only using a few classes
- System.Runtime.InteropServices
  - COM object identification
  - Parameter and return value marshalling
  - HRESULT behavior

COM Support
Example

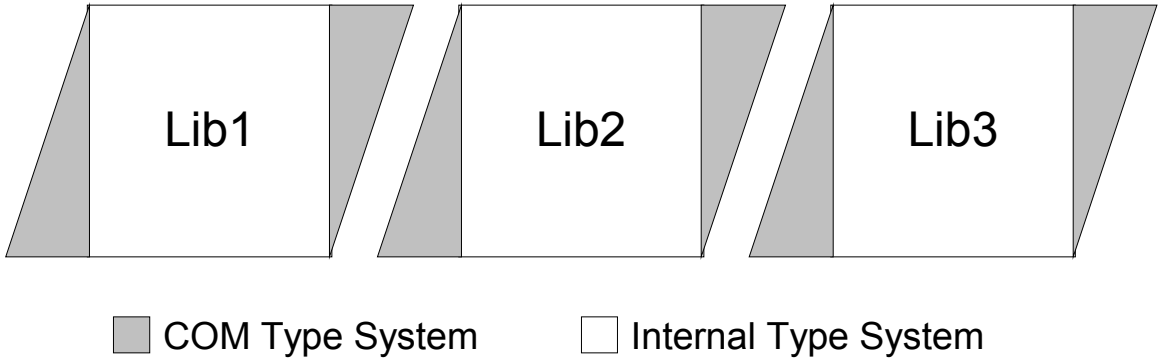
```
[Guid(“56A868B1-0AD4-11CE-B03A-0020AF0BA770”)]
interface IMediaControl
{
  void Run();
  void Pause();
  void Stop();
  …
  void RenderFile(string strFilename);
}
```

Back to projects

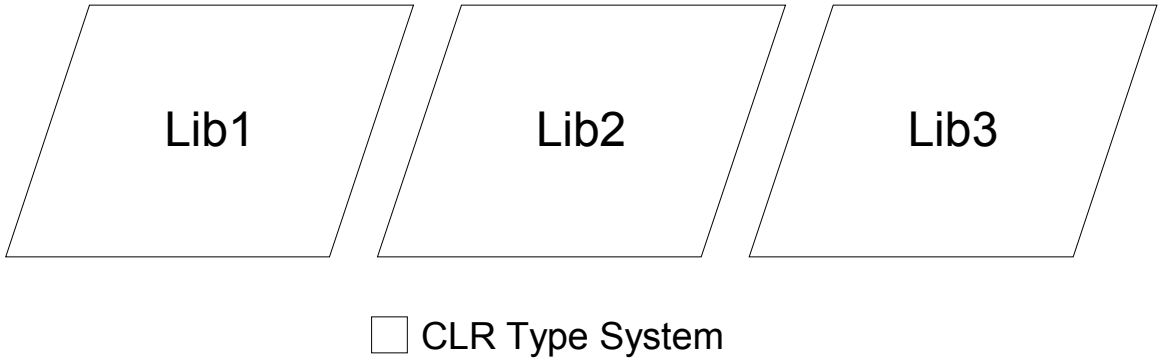
The Evolution of Components (Microsoft-style)



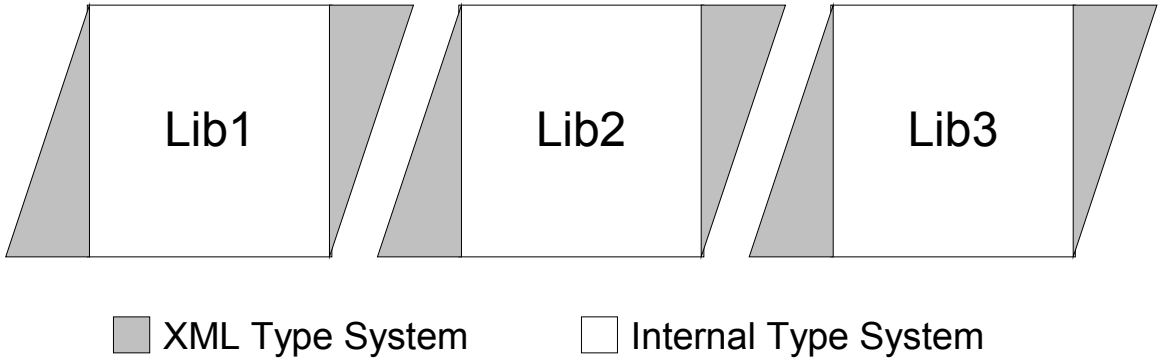
Type fragmentation under COM



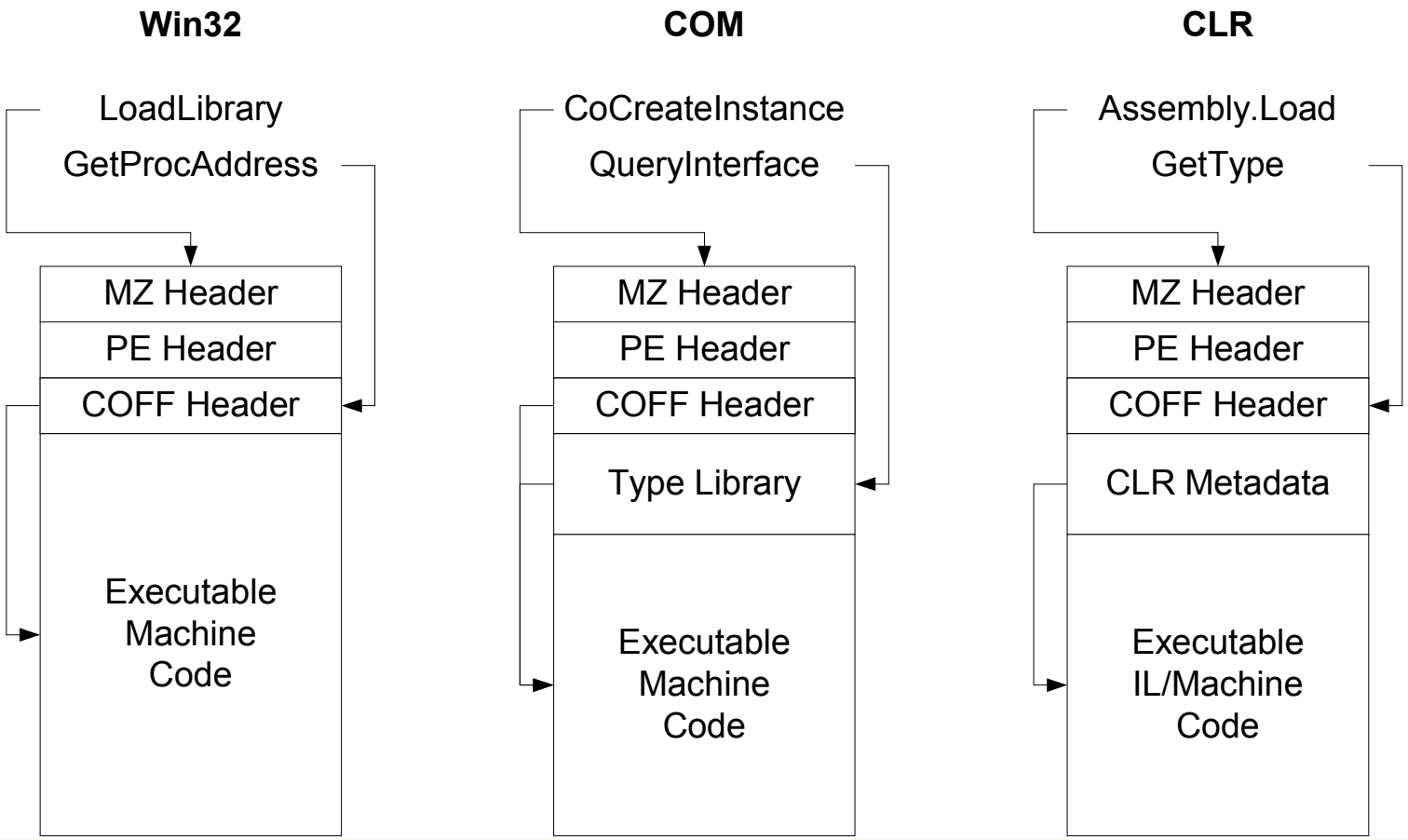
Pervasive Type in the CLR



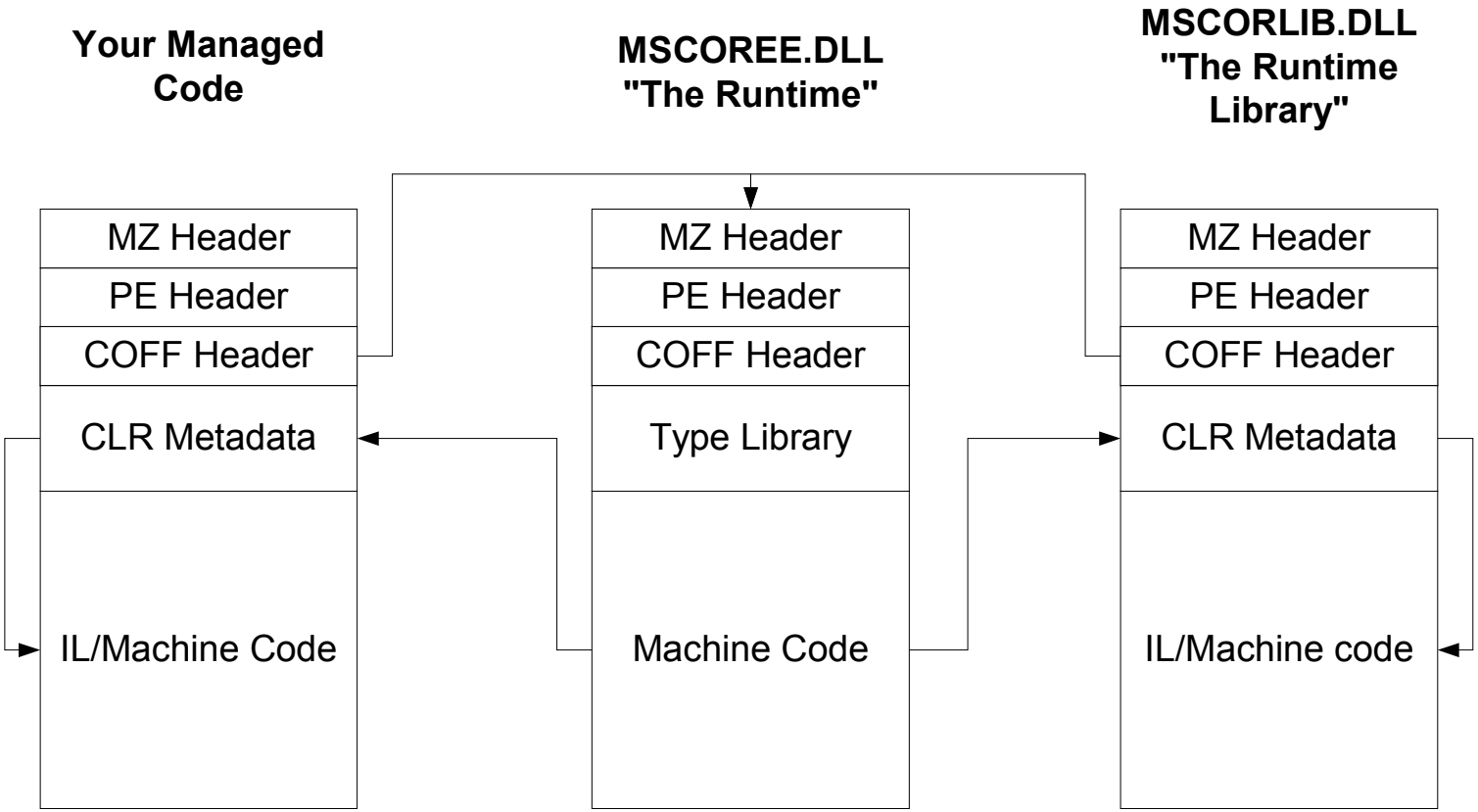
Type segregation under XML



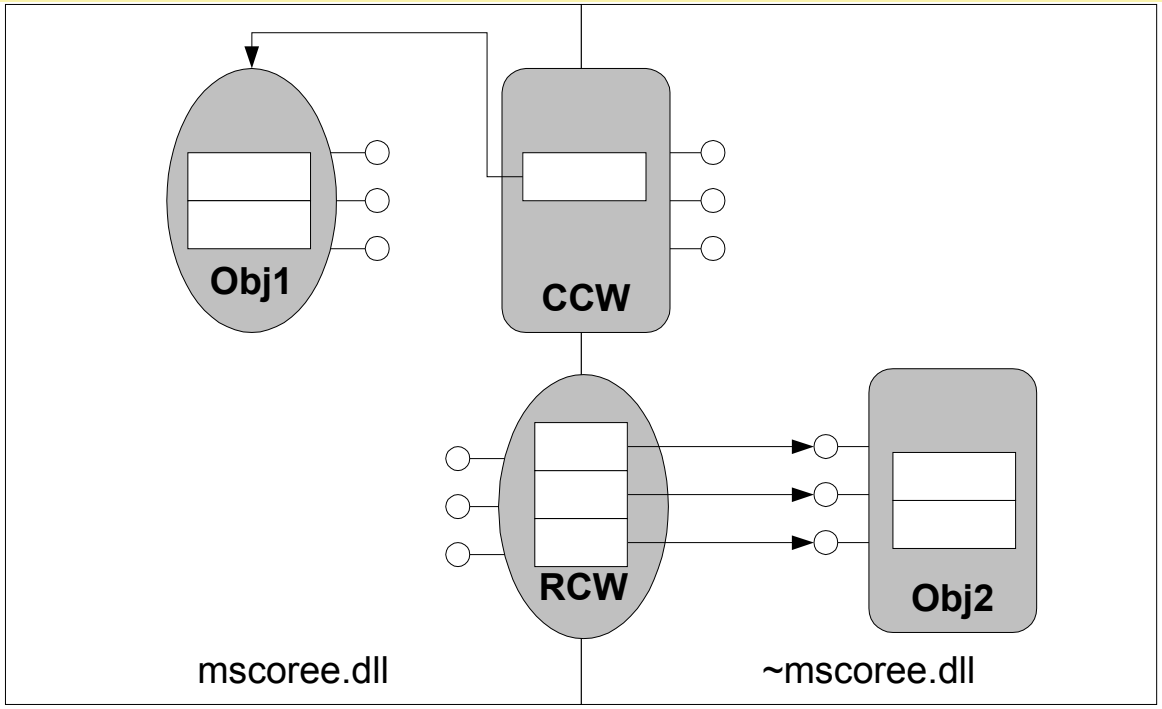
Object Creation and the Loader



CLR Architecture

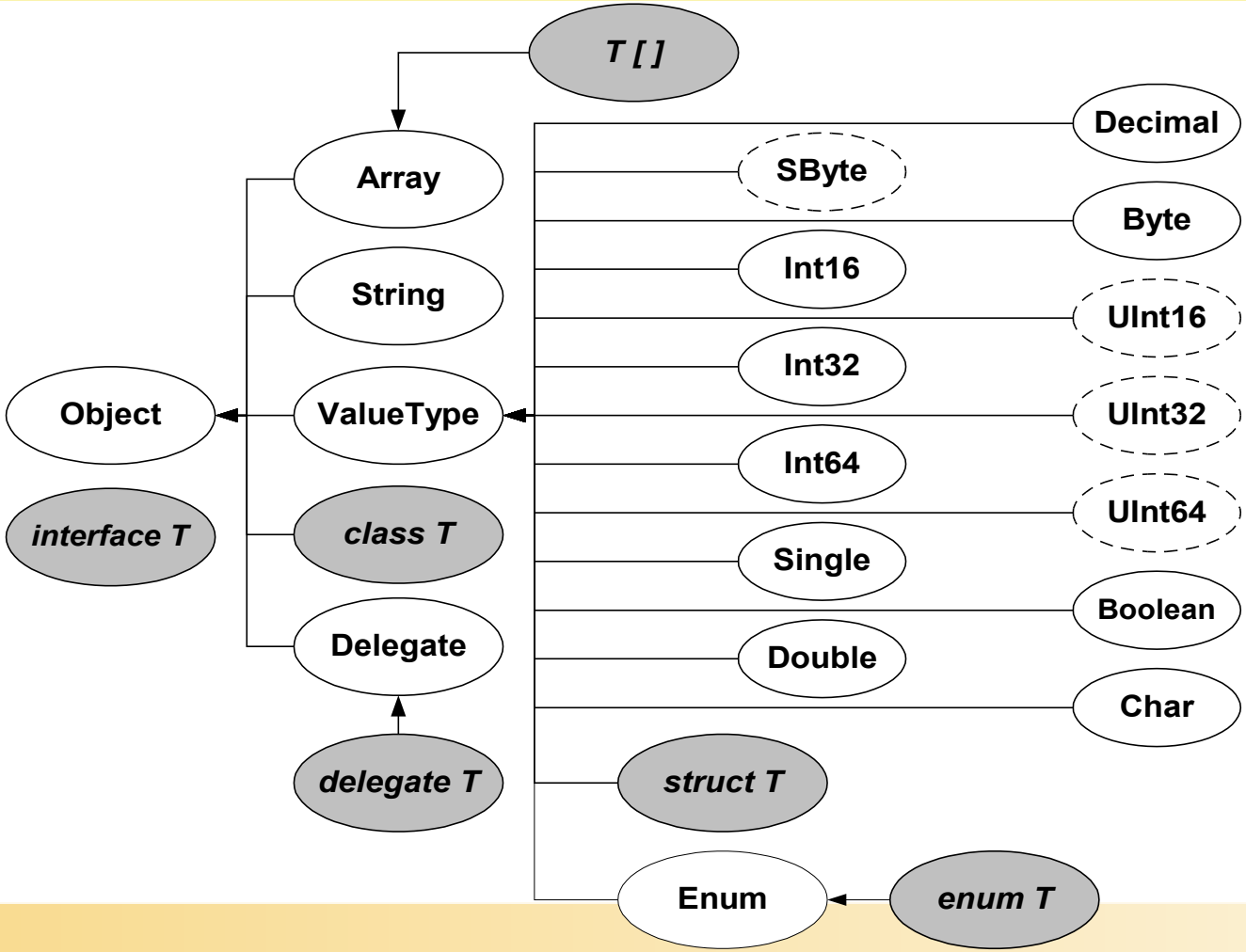


on-the-fly COM

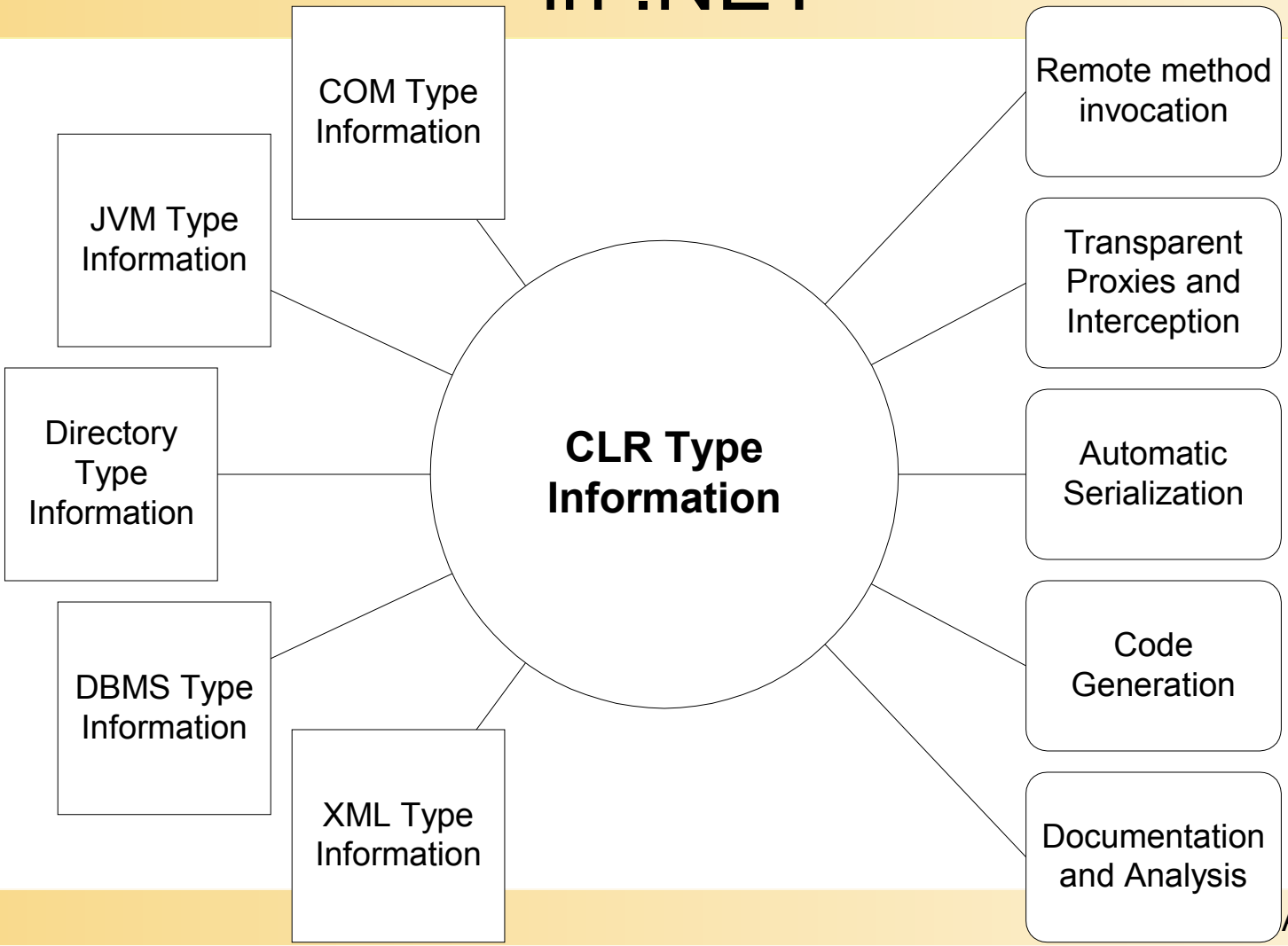


- COM callable wrapper (CCW)
- runtime callable wrapper (RCW)

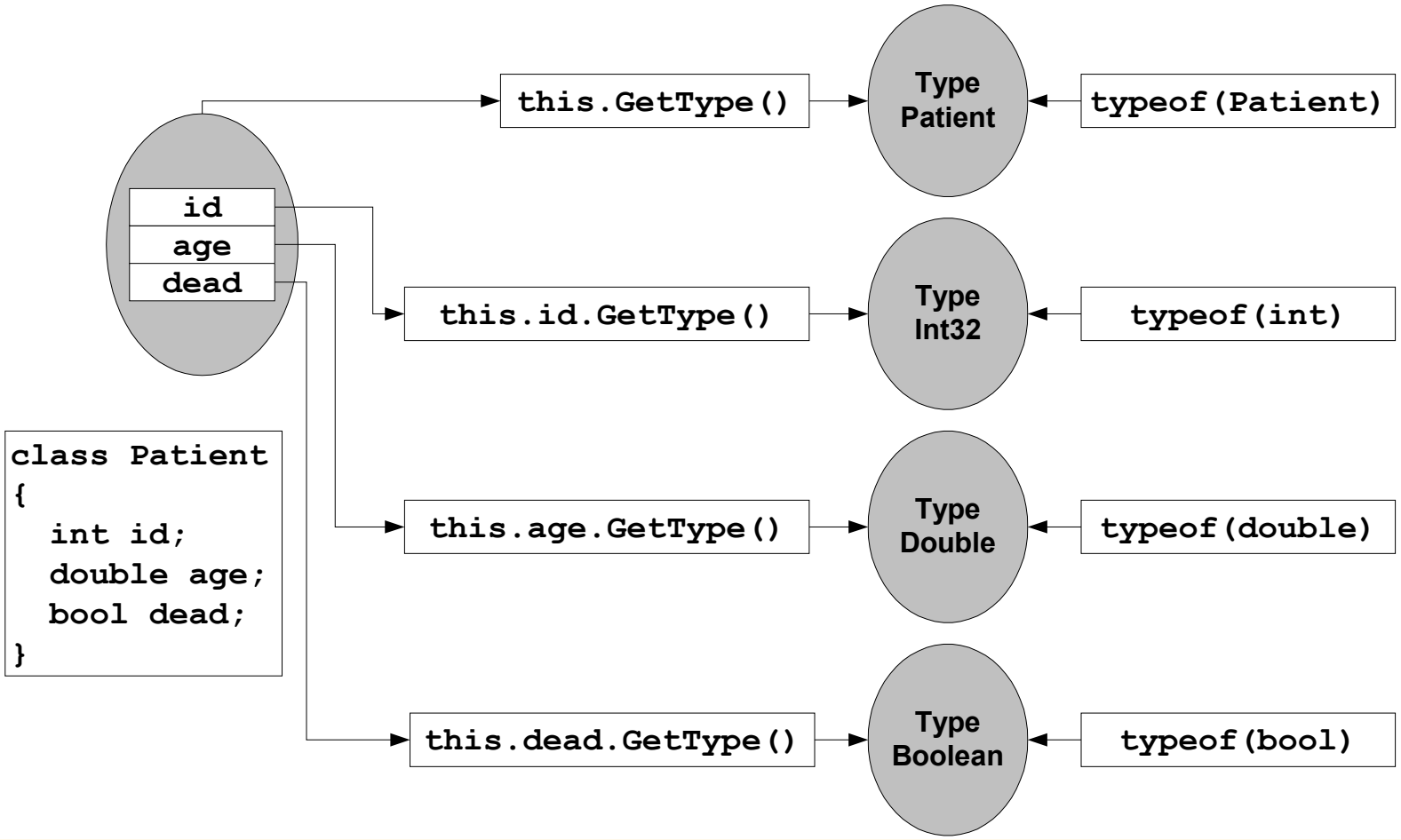
The CLR Type System



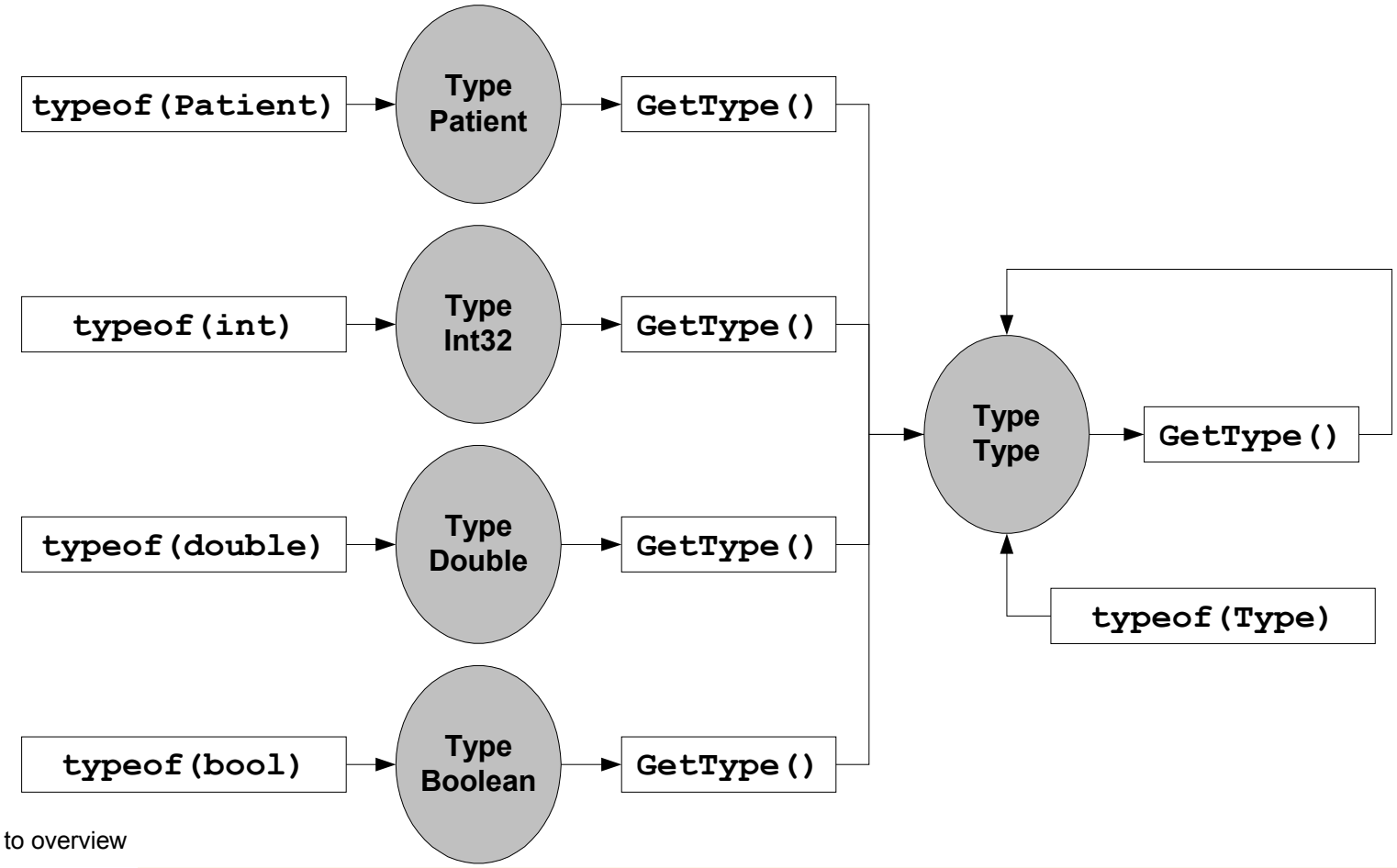
Metaprogramming and Reflection in .NET



Pervasive type and GetType



Pervasive type and System.Type



Back to overview