# 301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

***AP-20**: Components: the Microsoft way*

# Overview

- The Microsoft approach to components
- COM: Component Object Model
- The .NET framework
- Common Language Runtime
- .NET components
- Composition by aggregation and containment
- Communication by Events and Delegates

Chapter 15, sections 15.1, 15.2, 15.4, 15.11, and 15.12 of *Component Software: Beyond Object-Oriented Programming. C. Szyperski, D. Gruntz, S. Murer, Addison-Wesley, 2002*.

# Distributed Component Technologies

The goal:

- **Integration of services** for applications on various platforms
- **Interoperability**: let disparate systems communicate and share data seamlessly

Approaches:

- Microsoft: DDE, COM, OLE, OCX, DCOM and ActiveX
- Sun: JavaBeans, Enterprise JavaBeans, J2EE
- CORBA (Common Object Request Broker Architecture)
- Mozilla: XPCOM (Gecko functionality as components)
- SOAP (using XML)

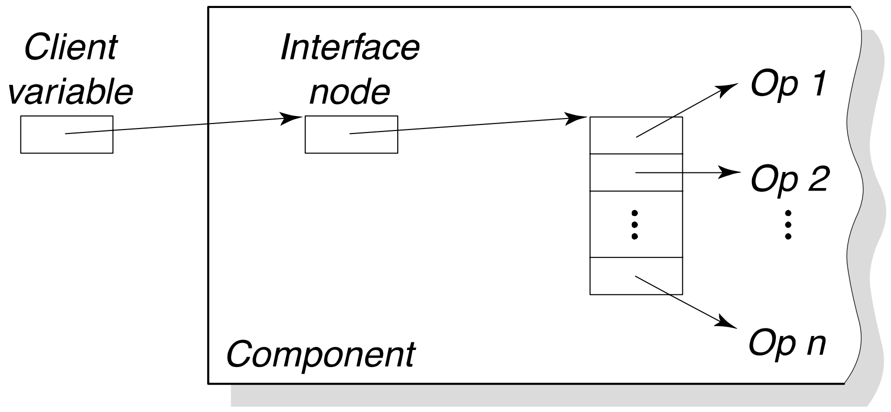
# The Microsoft Approach

- Continuous re-engineering of existing applications
- Component technology introduced gradually taking advantage of previous success, like
  - Visual Basic controls
  - Object linking and embedding (OLE)
  - Active X, ASP
- Solutions mainly adopted on MS platforms
- Review from main older approach (COM) to .NET + CLR

## COM: Component Object Model

- Underlying most MS component technologies (before .NET)
- Made available on other platforms, but with little success
- COM does not prescribe language, structure or implementation of an application
- COM only specifies an object model and programming requirements that enable COM components to interact
- COM is a **binary standard for interfaces**
- Only requirement: code is generated in a language that can create structures of pointers and, either explicitly or implicitly, call functions through pointers.
- Immediate for some languages (C++, SmallTalk) but possible for many others (C, Java, VBscript,…)

# COM interfaces and components



- A COM interface is a pointer to an interface node, which is a pointer to a table of function pointers (also called **vtable**)
- Note the double indirection

- **Invocation specification**: when an operation (**method**) of the interface is invoked, a pointer to the interface itself is passed as additional argument (like **self** or **this**)
  - The pointer can be used to access instance variables
- **COM component** may implement any number of interfaces.
- The entire implementation can be a defined in a single class, but it does not have to be.
- A component can contain many objects of different classes that collectively provide the implementation of the interfaces provided by the component.

# A COM component with 3 interfaces and 2 objects

- Object 1 implements Interfaces A and B,
- Object 2 implements Interface C
- Interfaces must be mutually reachable
- Possible according to COM specification, rare in practice

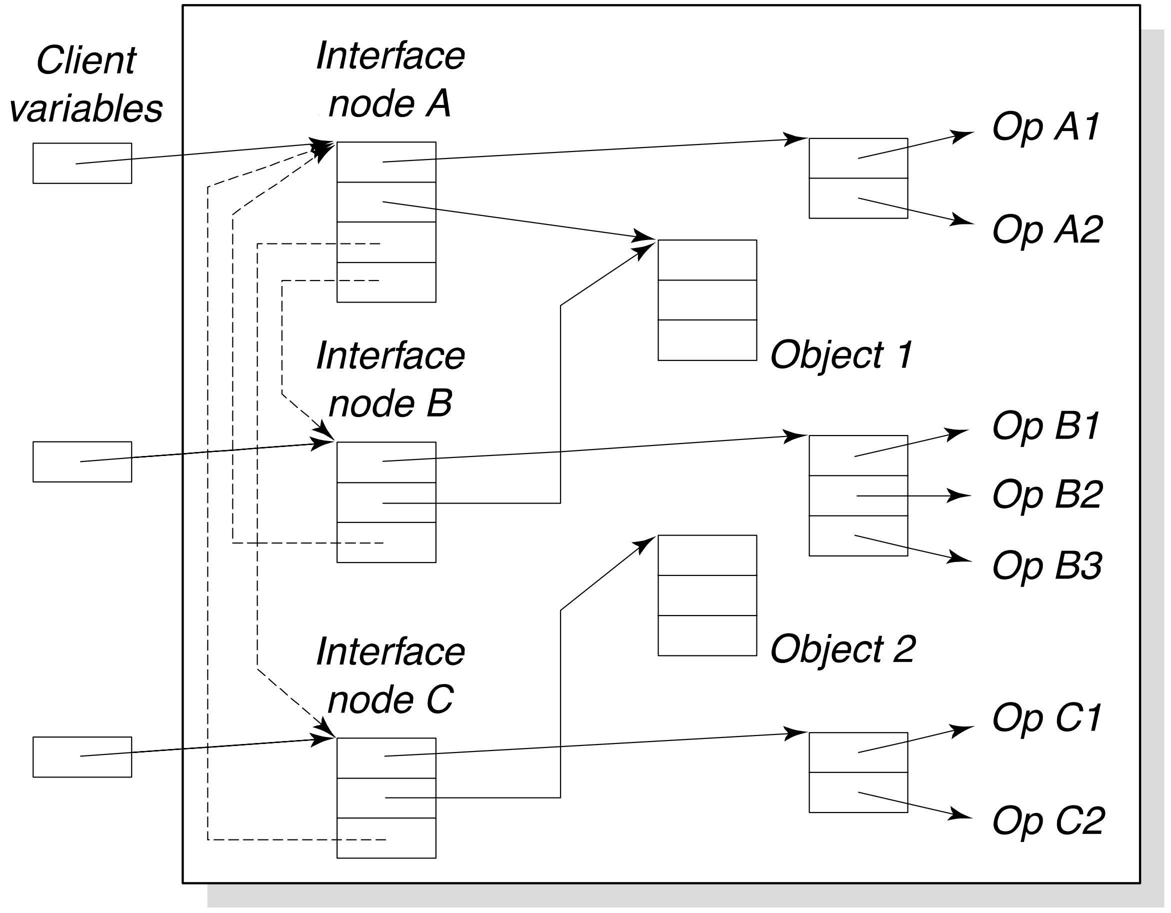
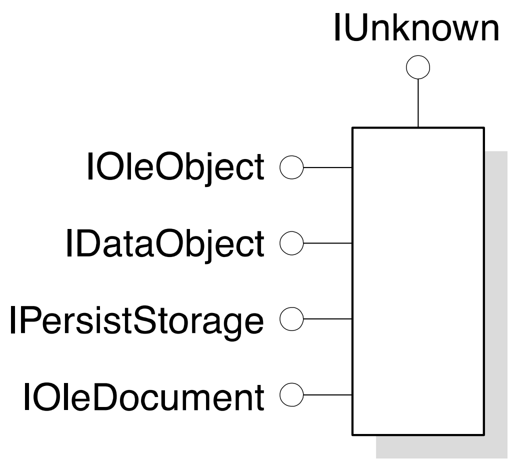


***Figure 15.2*** A COM object with multiple interfaces.

# COM Interfaces

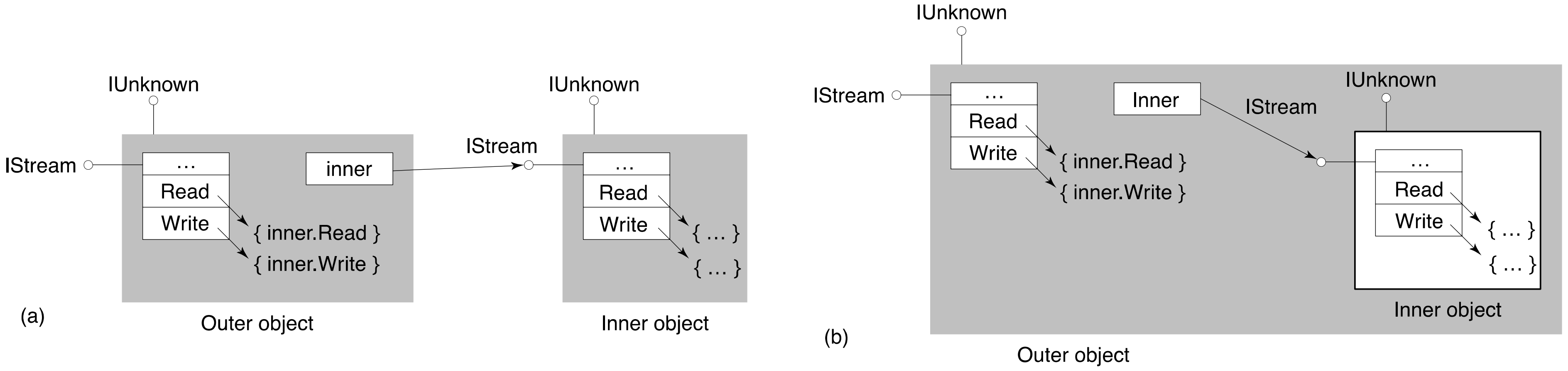


- Identity determined by **Globally unique identifiers (GUID)** (128 bits) or (non-unique) name
- **`IUnknown`**: root of interface hierarchy, includes:
  - **`QueryInterface`**
  - **AddRef** and **Release** (for Garbage Collection via Reference Counting)
- **`QueryInterface`** (GUID -> Interface reference/error) allows to know if an interface is implemented by the component
- "Invocations to **`QueryInterface`** argument **`IUnknown`** on the same component must return the same address"
- Thus **`IUnknown`** used to get the "identity" of a component

```
[ uuid(00000000-0000-0000-C000-000000000046) ] interface IUnknown {
HRESULT QueryInterface ([in] const IID iid, [out, iid_is(iid)] IUnknown iid);
unsigned long AddRef ();
unsigned long Release (); }
```

# COM component reuse: Containement

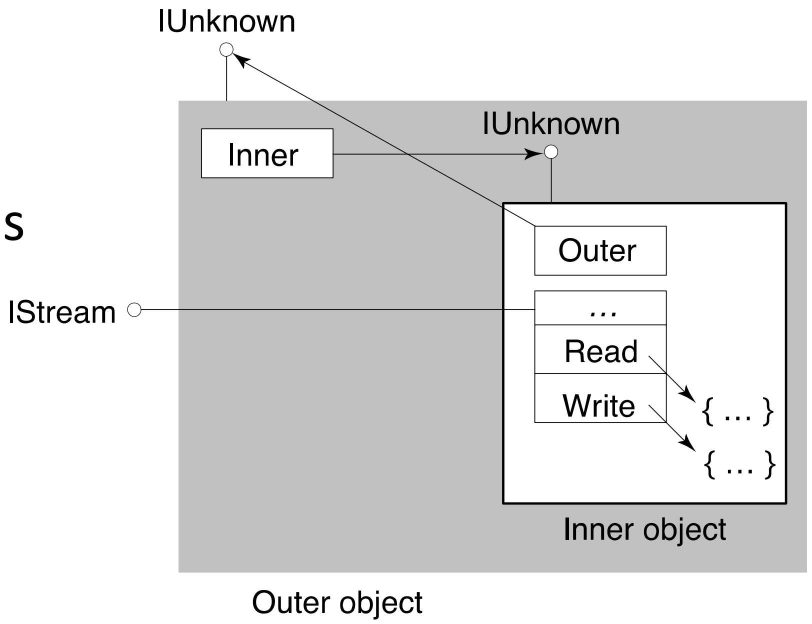
- COM does not support for implementation inheritance
- Reuse supported through Containement & Aggregation
- **Containement**: an outer objects holds an exclusive reference to an inner object
- Requests to outer can be forwarded to inner, simply invoking one of its methods



# COM component reuse: Aggregation

- Containement adds overhead for calling and returning from methods: could cause a performance issue
- With **aggregation**, a reference to the interface of Inner is passed to the client.
- Outer cannot intercept / modify / filter invocations to Inner
- Problem: The client should not be aware of the fact that Inner is serving instead of Outer (transparency)

- This can be achieved (only) with collaboration of the Inner object: calls to **QueryInterface** are forwarded to the **IUnknow** interface of Outer



## COM inheritance, polymorphism versioning

- Single inheritance among interface possible but rarely used (eg **IUnknown**, **IDispatch** and few others)
- But due to the **QueryInterface** mechanism impossible to know if an interface has more methods
- Polimorphism given by support to sets of interfaces for components:
  - The type of a component is the set of GUID of its interfaces
  - A subtype is a superset of interfaces
- COM does not support interface versioning

# Creating COM objects

- An application can request a COM component at runtime, based on its class
- Class identifiers are GUIDs (called CLSIDs)
- Procedural static API for creting objects:
  - **CoCreateInstance(CLASID, IID)**
- Exploits a registry to identify a (local or remote) COM server which provides a Factories for COM Interfaces

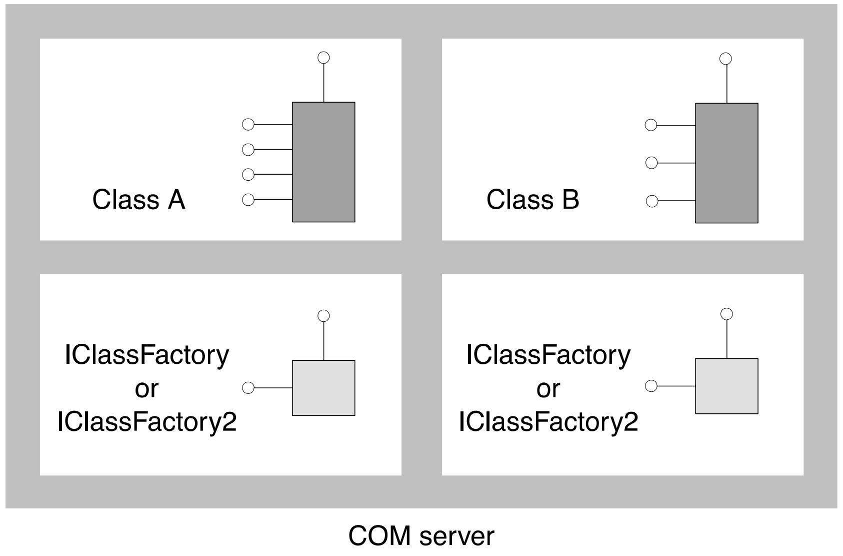


***Figure 15.9*** COM server with two coclasses, each with a factory.

# The .NET Framework: Summary

- The **.NET framework** and **.NET components**
- Types of .NET components, connections of components, and deployments
- Local and distributed components
- **Aggregation** and **containment** compositions
- **Synchronous** and **asynchronous** method invocations
- **Delegates** and Event-based communication

# The .NET Framework

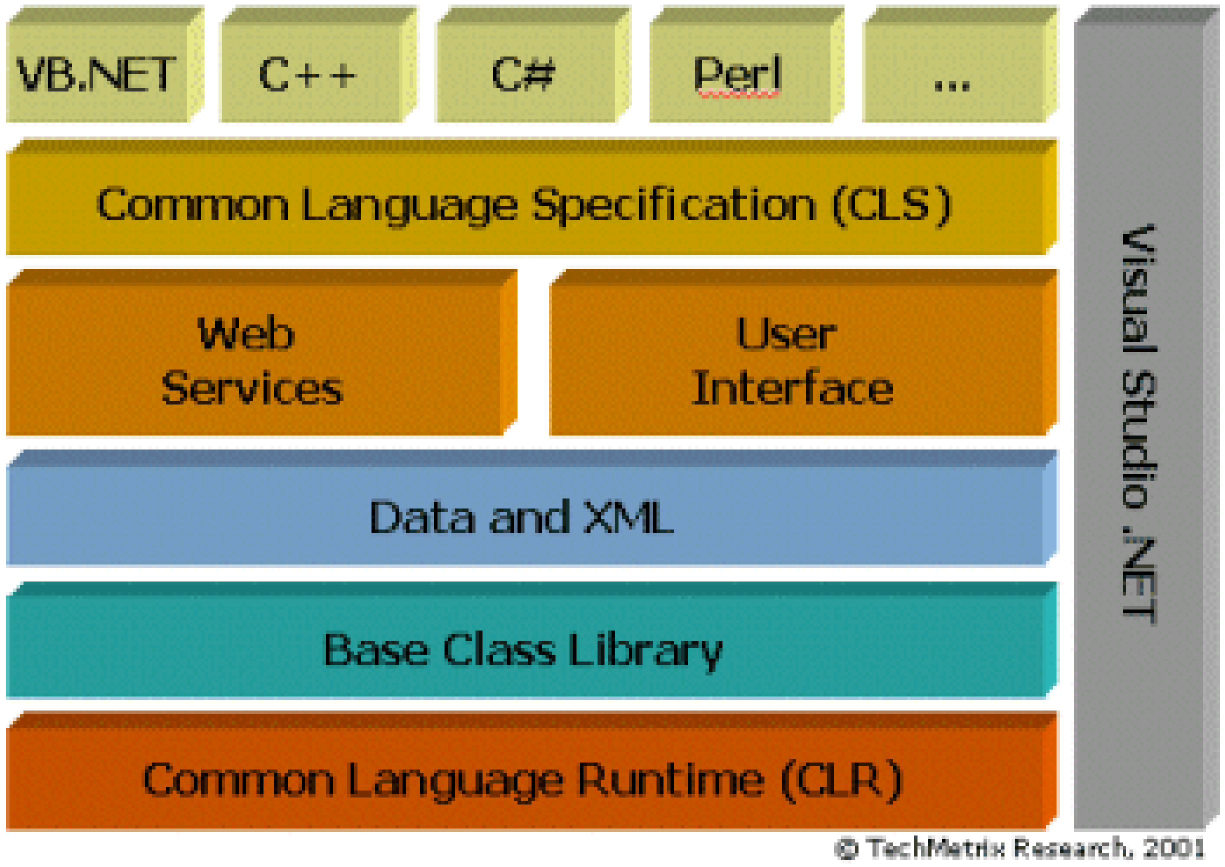
- Introduced by Microsoft in 2000.
- Platform for rapid and easier building, deploying, and running secured **.NET software components**
- Support for rapid development of **XML web services** and **applications**
- Highly productive, **component-based**, **multi-language** environment for integrating existing applications with internet
- Emphasis on **interoperability**

# The .NET Framework consists of:

- **The Common Language Specification (CLS)**
  It contains guidelines, that language should follow so that they can communicate with other .NET languages. It is also responsible for Type matching.
- **The Framework  Base Class Libraries (BCL)**
  A consistent, object-oriented library of prepackaged functionalities and Applications.
- **The Common Language Runtime (CLR)**
  A language-neutral development & execution environment that provides common runtime  for application execution .

# .NET Framework structure

(http://www.dotnet101.com/articles/art014_dotnet.asp)



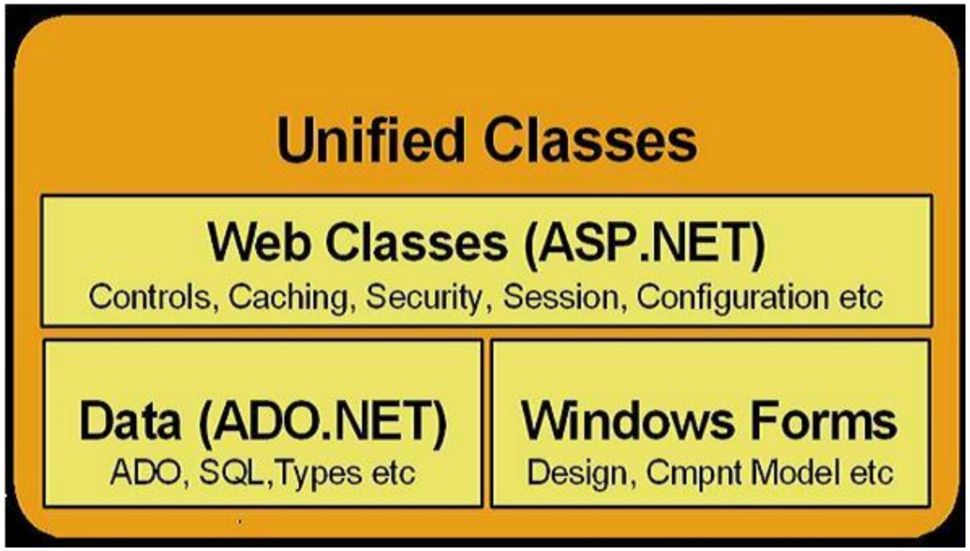
# Common Language Specification

**CLS performs the following functions:**

- Establishes a framework that helps enable cross-language integration, type safety, and high performance code execution
- Provides an object-oriented model that supports the complete implementation of many programming languages
- Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other

# .NET Framework Base Class Library

- The Class Library is a comprehensive, object-oriented collection of reusable types
- These class library can be used to develop applications that include:
  - Traditional command-line applications
  - Graphical user interface (GUI) applications
  - Applications based on the latest innovations provided by ASP.NET
    - Web Forms
    - XML Web services



# Common Language Runtime (CLR)

- CLR ensures:
  - A common *runtime* environment for all .NET languages
  - Uses *Common Type System (strict-type & code-verification)*
  - Memory allocation and garbage collection
  - Intermediate Language (MSIL) to native code compiler.
  - Security and interoperability of the code with other languages
- Over 36 languages supported today
  - C#, VB, Jscript, Visual C++ from Microsoft
  - Perl, Python, Smalltalk, Cobol, Haskell, Mercury, Eiffel, Oberon, Oz, Pascal, APL, CAML, Scheme, etc.

# Overview of .NET Framework (cont.)

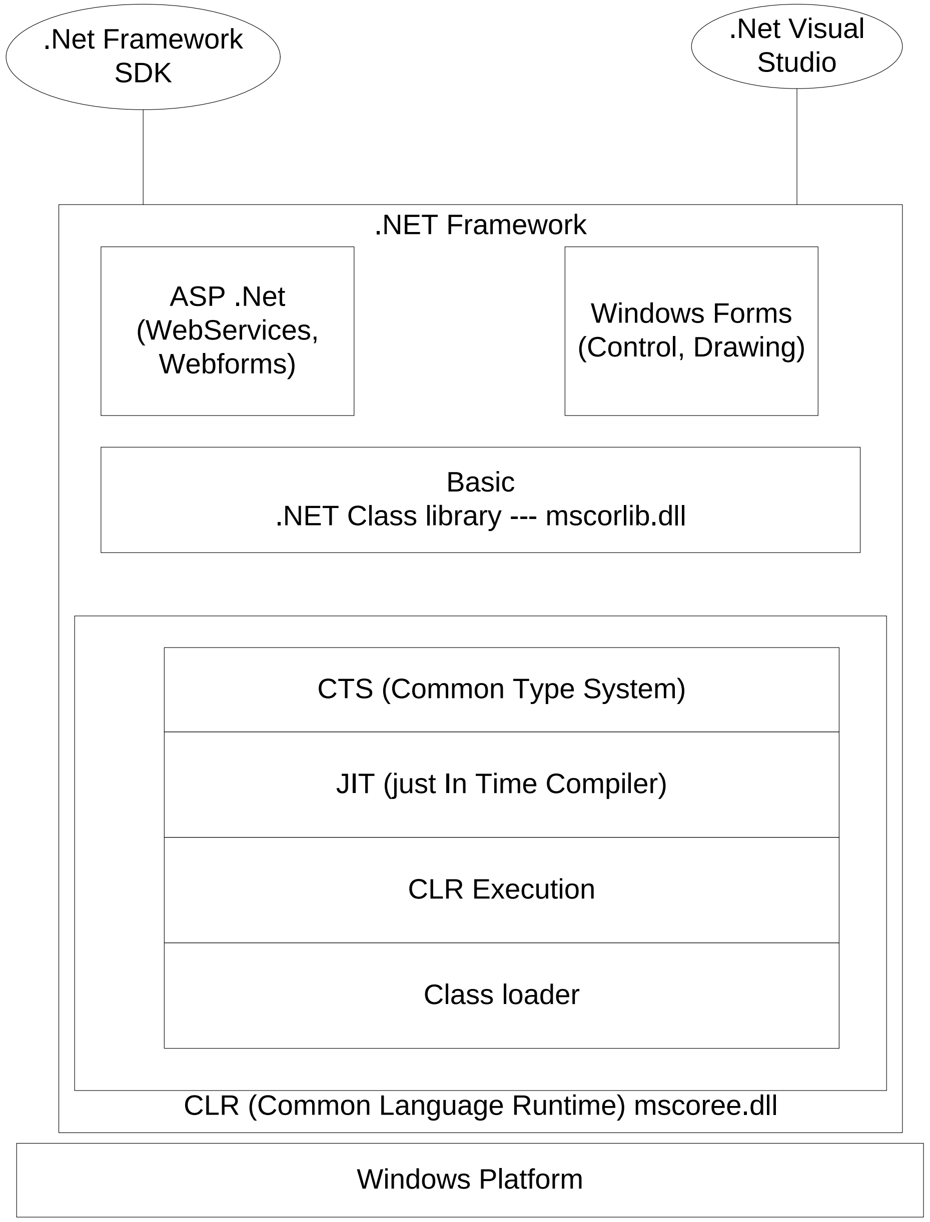
- Supports development and deployment of **desktop**, **window**, and **web-based application services** on both Windows platforms and on other platforms through **SOAP** and **HTTP**
- .NET simplifies and improves **support for components development and deployment** w.r.t. Component Object Model (COM), and  Distributed COM (DCOM) technology.
- COM components can be reused. Differently from COM, .NET technology supports **component versions**, and different versions can coexist without any conflict.

## Overview of .NET Framework (cont.)

- Support of **distributed components** by **Remoting Channel technology**.
- Supports of **Interoperability** between COM, .NET and XML web service components.
- The .NET framework is available in **.NET Framework SDK** and **Visual Studio.NET IDE SDK** which support writing, building, testing, and deploying of .NET applications.
- It supports all **.NET languages** such as VB.NET, VC.NET, C#, and many others.

## Microsoft CLI (Common Language Infrastructure): some historical notes

- When Java became popular Microsoft joined the initiative
- The idea was to exploit the dynamic load features of JVM to implement a component based architecture like COM
- There were two main problems:
  - Interoperability with the existing code (COM)
  - Support for many programming languages
- Microsoft extended the JVM but Sun complained of license infringement
- Microsoft started developing its own technology
- This was based on their experience on Java, but they tried to address the two problems above
- The result was the **Common Language Infrastructure** (CLI)
- The core of CLI is the **Common Language Runtime** (CLR) which plays the same role as the JVM in Java


.Net Framework SDK
.Net Visual Studio
.NET Framework
ASP .Net (WebServices, Webforms)
Windows Forms (Control, Drawing)
Basic .NET Class library --- mscorlib.dll
CTS (Common Type System)
JIT (just In Time Compiler)
CLR Execution
Class loader
CLR (Common Language Runtime) mscoree.dll
Windows Platform

# Common Language Infrastructure

## Some CLI implementation

- CLR – Microsoft’s commercial offering
- SSCLI ( code-named “Rotor” ) – Microsoft’s Shared Source CLI (free, but not for commercial use; discontinued)
- Mono - open source project initiative sponsored by Ximian ( now a part of Novell ) and now by Microsoft
- DotGNU Portable .NET – till ~2008
- OCL – portions of the implementation of the CLI by Intel – till ~2002
- .Net Core – Open Source, Cross-platform, Supported by Microsoft and community (V3.1 Released 2019-12-03)
- .Net  5.0.5 – Released 2021-04-06

# Common features of CLR and JVM

- Secure
- Portable
- Automatic MM (GC)
- Type safety
- Dynamic loading
- Class Library
- OOP
- Mix-in inheritance

Note that the essential traits of the execution environment are similar, though there are relevant difference in the design

CLI has been standardized (ECMA and ISO) and is a superset of Java. We will refer mainly to CLR.

# Foundation of .NET framework – CLR

- **Common Language Runtime (CLR)** is a virtual machine environment sitting on the top of the operating system.
- CLR consists of **Common Type System (CTS)**, **Just-In-Time CIL Compiler (JIT)**, **Virtual Execution System**, plus other management services (**garbage collection**, **security management**).
- **CLR** is like **JVM** in Java. It is assembled in a package of assembly consisting of **MS Intermediate Language (MSIL)** code and **manifest** (Metadata about this packet).
- The **CIL** code is translated into native code by JIT compiler in CLR. IL code **is verified by CTS** first to check the validity of data type used in the code.

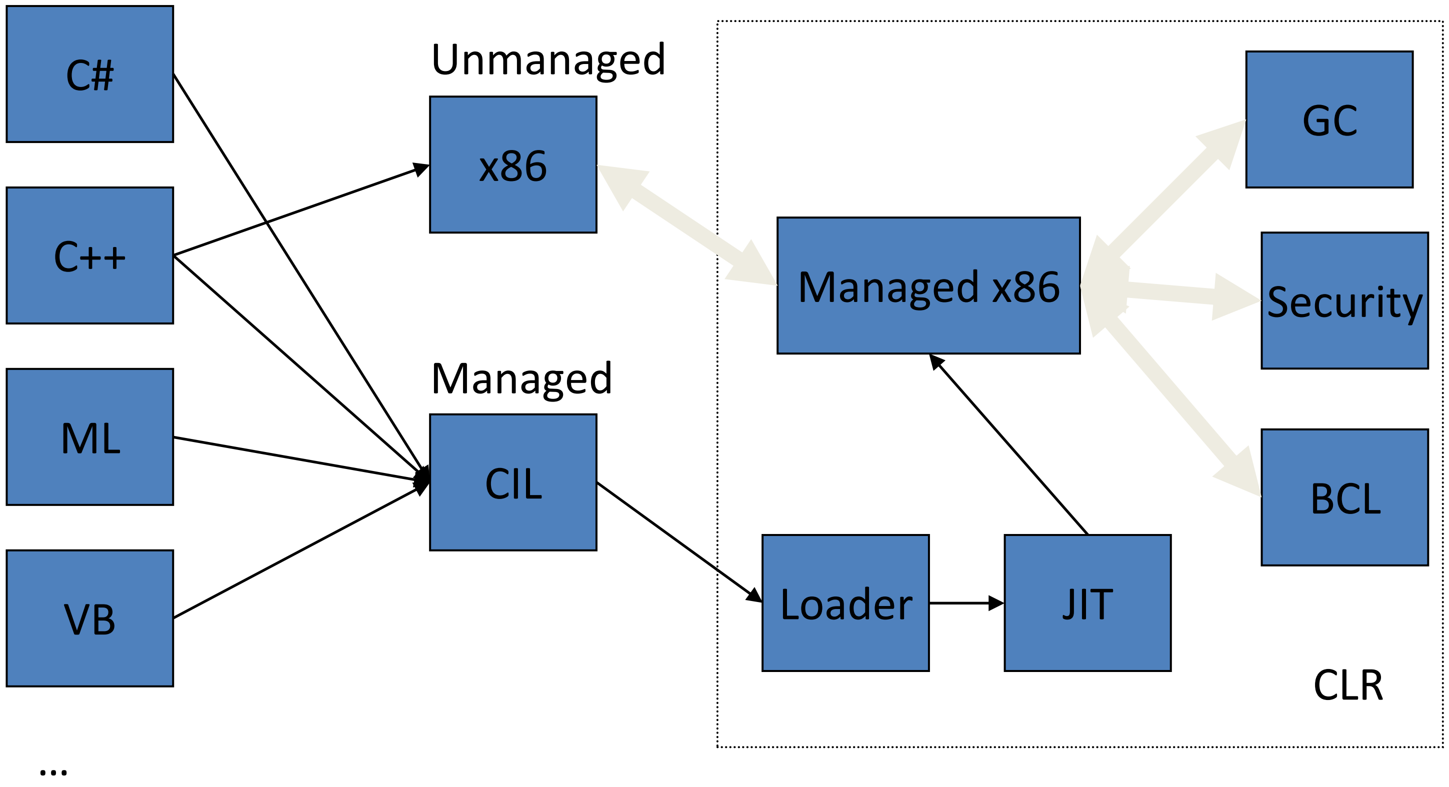
# Foundation of .NET framework – CLR

- **Multilanguage support**: (VB, managed C++, C# etc) by Common Language CLR implementation.
- A class in one language can inherit properties and methods from related classes in other languages.
- The **CTS** defines a standard set of data type and rules for creating new types.
  - Reference types
  - Value types
- The code targeting CLR and to be executed by CLR is called **.NET *managed* code**. All MS language compilers generate managed codes that conform to the CTS.

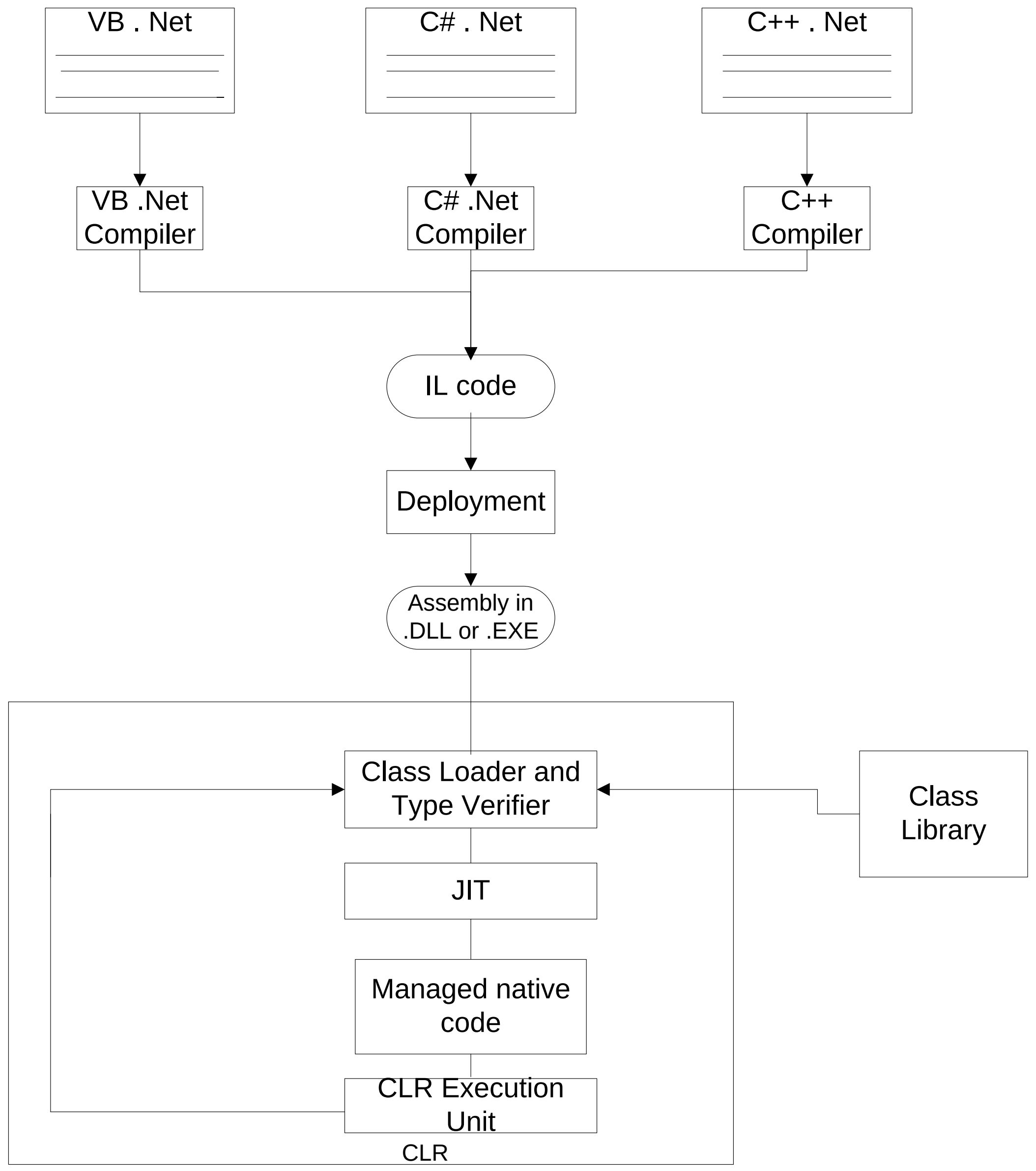
# Foundation of .NET framework – CLR

- The **CIL** code is like Java byte code. Regardless of the source programming languages, IL codes can interact by support of the CLR.
- The IL code can be in the format of **executable (.EXE)** or **Dynamic Link Library (.DLL)**.
- If these IL codes are generated by .NET compiler, they are called ***managed code***.
- The managed code can be executed only on **.NET aware platform**. Some DLL or EXE generated by non .NET compilers (such as early version of VC++) are called ***un-managed*** code.

# How CLR works



# Foundation of .NET framework – CLR

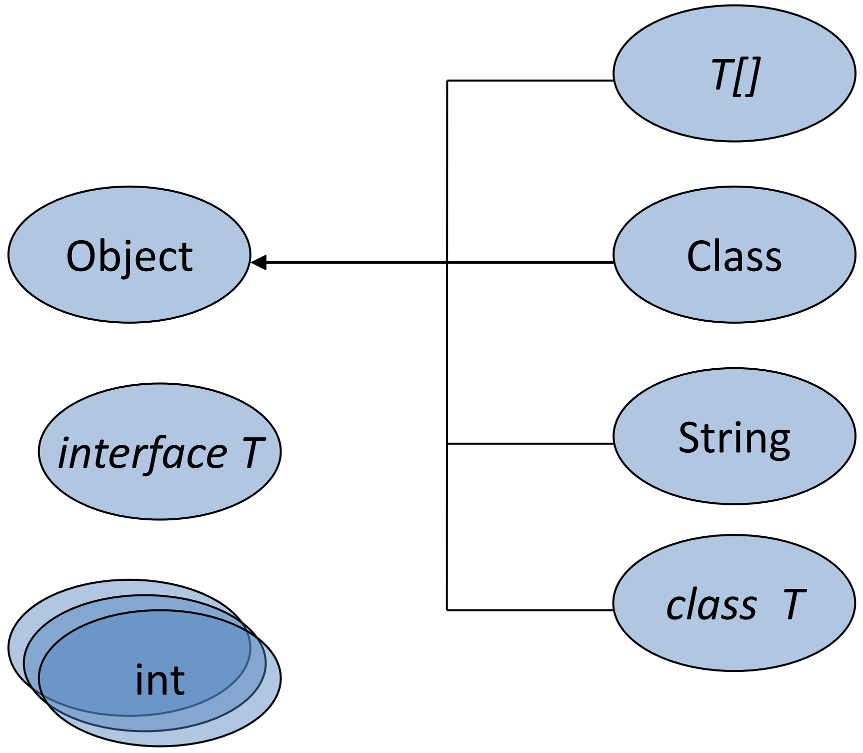


# Towards the CommonType System

- Execution environments such as CLR and JVM are ***data oriented***
- A type is the unit of code managed by the runtime: loading, code, state and permissions are defined in terms of types
- Applications are set of types that interact together
- One type exposes a static method (Main) which is the entry point of the application: it loads the needed types and creates the appropriate instances

# Java type system

- There are base types: **primitive types**, **Object**, **String** and **Class** (which is the entry-point for reflection)
- Type constructors are:
  - **Array**
  - **Class**
- The primitive types are unrelated to Object with respect to inheritance relation
- This applies to interfaces too, but objects that implements interfaces always inherit from Object
- Java type system is far simpler than the one of CLR

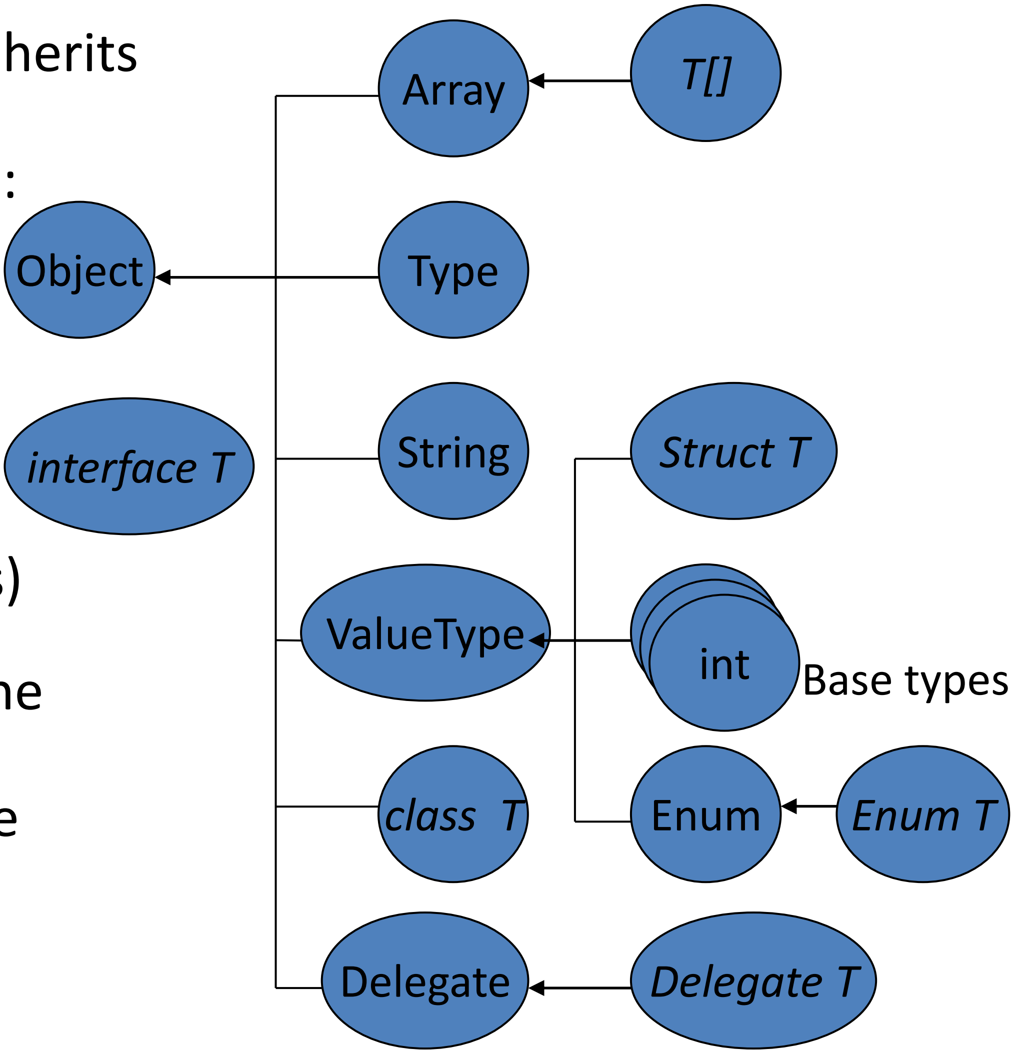


# Common Type System

- Goal: To establish a framework to support cross-language interoperability, type safety, and high performance code execution.
- Defines a rich set of data types, based on an object-oriented model.
- Defines rules that ensure that objects written in different languages can interact with each other.
- Specifies the rules for scopes, type visibility and access to the members of a type. The Common Language Runtime enforces the visibility rules.
- Defines rules of type inheritance, virtual methods and object lifetime.
- Languages supported by .NET can implement only part of the common data types.

# CLR Common Type System

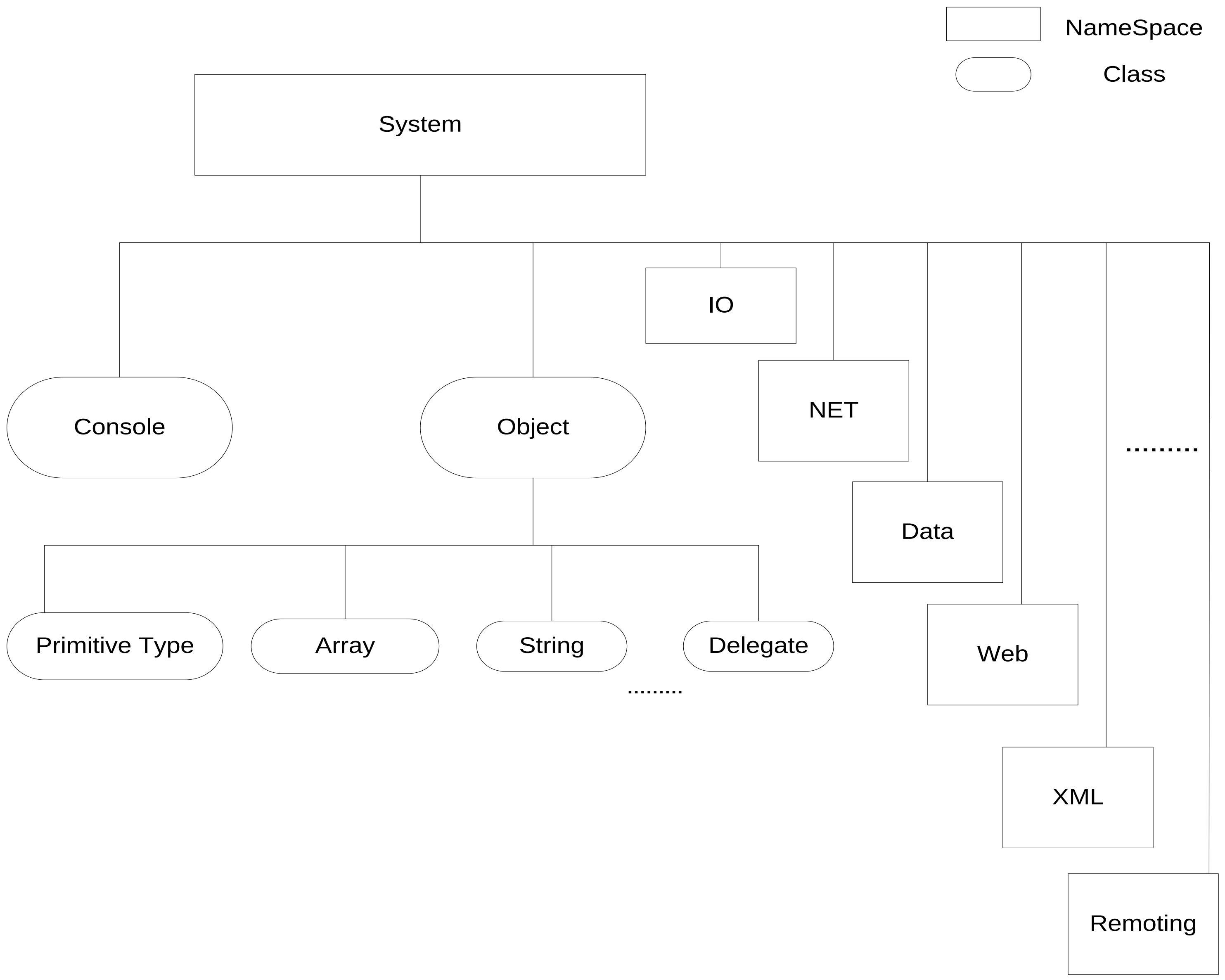
- Common rooted: also numbers inherits from **Object**
- There are more type constructors:
  - **Enum**: constants
  - **Struct**: like class but without inheritance and stack allocation
  - **Delegate**: type that describes a set of methods with common signature
- **Value types** (numbers and structs) inherits from **Object**. Still are not references and aren't stored on the heap
- The trick is that when a value type should be upcasted to **Object** it is ***boxed*** in a wrapper on the heap
- The opposite operation is called ***unboxing***



# The .NET Framework Class Library

- The .NET framework **class library** is a collection of reusable basic classes which are well organized by **namespaces**.
- Correspond to **Java API** and **packages**
- A namespace consists of many classes and sub-namespaces. It is deployed as a **component class library** itself and is organized in a component–based hierarchy.
- The .NET framework itself is built up in a component model.
- Developers can create **custom namespaces**
- A namespace can be deployed as an **assembly** of binary components.
- **`using <namespace>`** in C# or
  **`import <namespace>`** in VB
  to access classes in a namespace.

# The .NET Framework Class Library



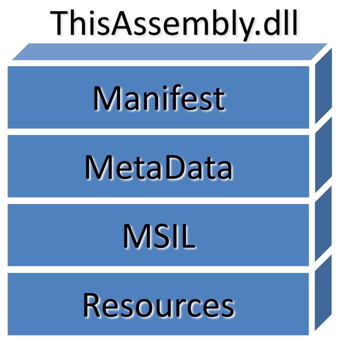
# The Component Model of .NET

- **Assemblies** (or **CIL DLL components**) replace the COM Components
- The .NET component technology is **unified-language oriented**. Any .NET component is in the format of **pre-compiled MSIL**, which can be binary plugged in by any other MSIL components or any other .NET compatible clients.
- A **.NET component** is a single **pre-compiled** and **self described CIL module** built from one or more classes or multiple modules deployed in a **DLL assembly file**.

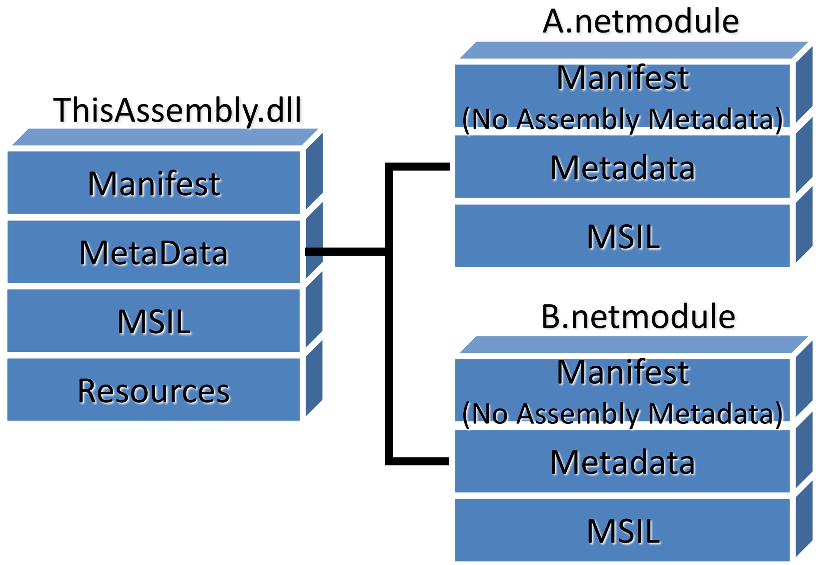
# Assemblies

- **Assemblies** are the smallest unit of code distribution, deployment and versioning
- Individual components are packaged into assemblies
- Can be **dynamically loaded** into the execution engine on demand either from local disk, across network, or even created on-the-fly under program control

*Single File Assembly*



*Multi File Assembly*



## Assembly characteristics

- Self-describing
  - To enable data-driven execution
- Platform-independent
- Bounded by name
  - Locate assemblies by querying its **strong name**
  - Strong name = (publisher token, assembly name, version vector, culture)
  - Version vector = (major, minor, build, patch)
- Assembly loading is sensitive to version and policy
  - Assemblies are loaded using tunable binding rules, which allow programmers and administrators to contribute policy to assembly-loading behavior.
- Validated
  - Each time an assembly is loaded, it is subject to a series of checks to ensure the assembly’s integrity.

# Assembly structure

- An assembly consists of up to four parts:
  1. **Manifest** (table of info records): name, version vector = (major, minor, build, patch), culture, strong name (public key from the publisher), type reference information (for exported types) list of files in the assembly, information on referenced assemblies reference.
  2. **Metadata** of modules
  3. **CIL code** of module
  4. **Resources** such as image files.

# The Component Model of .NET

- A **module** has **CIL** code and its metadata **but without manifest**. Not loadable dynamically. Building block at compile time to build up **an assembly Module file**. Extension: .**netmodule**.
- An **Assembly** is made up by one or many classes in a module. Assembly has a **manifest file** to self-describe the component itself.
- An assembly has a file extension .dll or .exe and is **dynamically loadable**.

# The Component Model of .NET

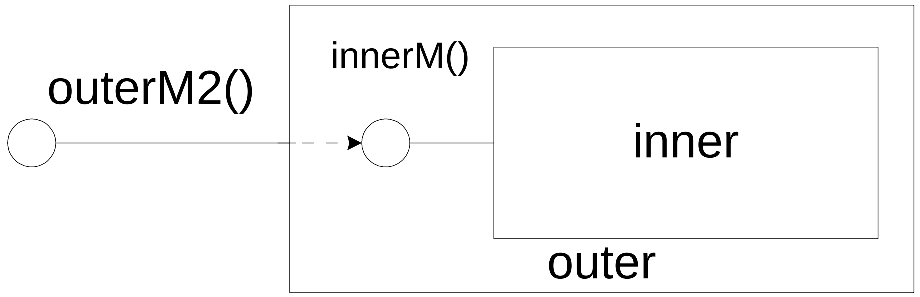
- A **.dll** file is not executable just like a class file is a byte code file that is not executable.
- An **.exe** file, generated by a .NET compiler, has a **PE .NET** format
  - PE (Portable Executable) is the standard MS format for executable files
  - PE .NET identifies the executable as for execution on the CLR: it causes a call to the CLR runtime at the beginning

# The Component Model of .NET

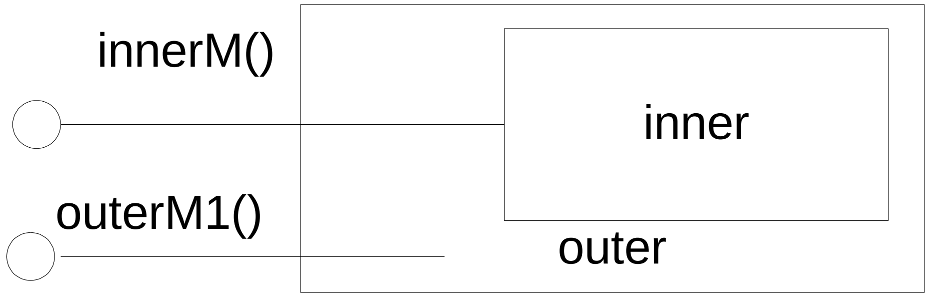
- A .NET component can be
  - **Local** (.dll), can only be accessed locally (within same application domain), in same machine
  - **Remote (distributed)** (.exe), can be accessed remotely in same machine or different machines.
- A .NET DLL component can be deployed
  - as a **private component**, knowing the target client
  - as a **shared public component**
- In the latter case it must be published (registered) in a centralized repository **Global Assembly Cache (GAC)**, typically using its strong name.
- A shared component supports side-by-side multiple version component execution.

## The Connection Model of .NET

- **.NET** component compositions enable the component **reuse** in either **aggregation compositions** or **containment compositions**.



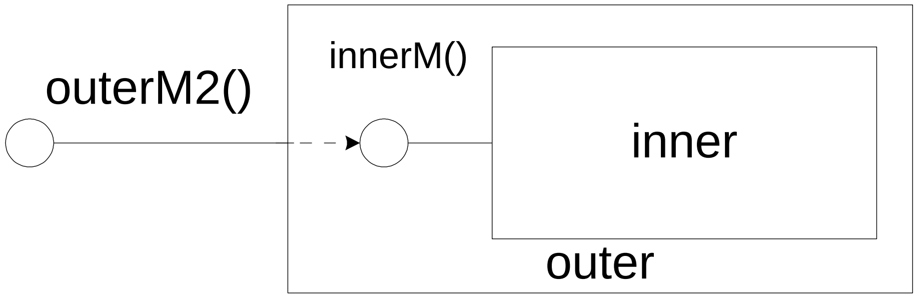
**containment**



**aggregation**

## Containment compositions

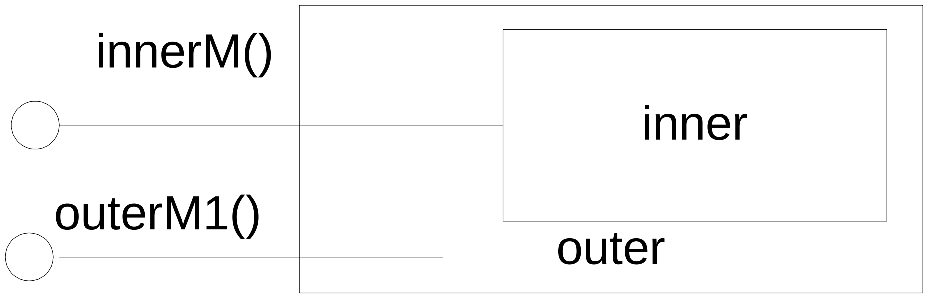
- If a request to the outer component needs help from an inner component the request is forwarded to that inner component.
- The outer component does not expose the interface of the inner component.
- The client is blind of the handler of the request. The **outerM2()** delegates a request to the **innerM()** method of inner component



**containment**

# Aggregation compositions

- The service of inner component hands out its service directly to the client of outer component.
  - The outer component exposes the interfaces of inner component.
  - The **innerM()** method of inner component becomes part of interface to the outer component



**aggregation**

- A .NET component can also be composed by mixed aggregations and containments in a flat structure or nested compositions in multiple levels in depth.

## DELEGATES IN CLR / C#

## What are Delegates?

- A **Delegate** (in CLR / C#) is a type that represents references to methods with a specific parameter list and return type.
- Eg: `delegate int MyFun(int i, int j);`
  is a type with instances holding methods of type
  `(int*int → int)`
- Similar to **function pointers in C++**, but **type-safe and secure**.
- An instance of a Delegate type can hold/refer both to static and to instance methods (of the prescribed signature).
- The method referred to by a delegate instance can be invoked by passing the list of actual parameters to the instance itself.

## Possibile uses of delegates

- Delegates can used to pass methods as arguments to other methods, thus supporting a functional programming style with some higher-order features.
- Delegates can be used to support event based programming, where event handlers are invoked through delegates.
- The ability to refer to a method as a parameter makes delegates ideal for defining callback methods.

## Example: use of delegate type in C#

```
class Foo {
delegate int MyFun(int i, int j);
static int Add(int i, int j)
   {return i + j;}
int Mult(int x, int y)
   {return x * y;}

static void Main(string[] args) {
   MyFun fun = new MyFun(Foo.Add);
   Console.WriteLine(fun(2, 3));
   Foo obj = new Foo();
   fun = new MyFun(obj.Mult);
   Console.WriteLine(fun(2, 3));
 }
}
```

- `type def MyFun instances:`
  `int * int -> int`
- `Delegate instance: static method`
- `Prints 5`
- `Delegate instance: instance method`
- `Prints 6`
- Note that **fun** must remember also obj

## Delegates like closures?

- In functional programming it is possible to define a function that refers to external variables
- The behavior of the function depends on those external values and may change
- **Closures** are used in functional programming to close open terms in functions
- Delegates are *not* equivalent to closures although they are a pair (env, func): the environment should be of the same type (class) to which the method belongs

# Functional programming in C#?

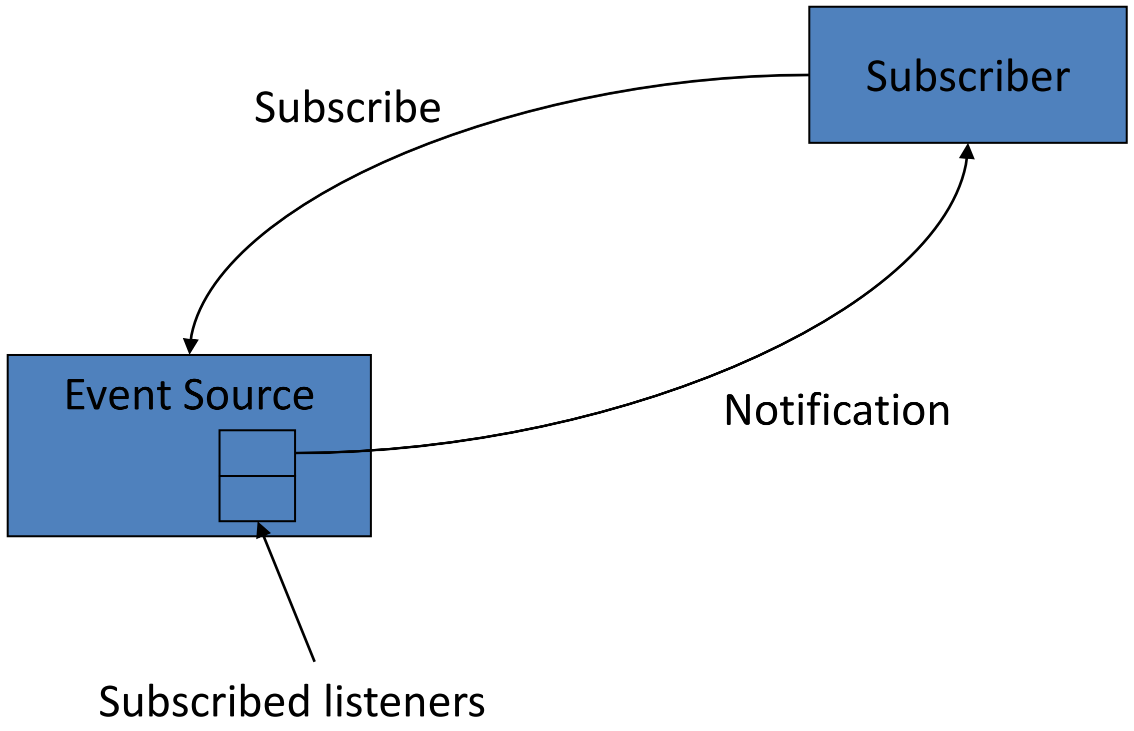
- Delegates allow representing static and instance methods as values, that can be passed as arguments
- Introduce elements of FP style in the mainstream, cleaner event model (call-backs can be naturally expressed as delegates)
- Example: mapping on an array:

```
delegate int MyFun(int);
int[] ApplyInt(MyFun f, int[] a) {
  int[] r = new int[a.Length];
  for (int i = 0;i < a.Length;i++)
    r[i] = f(a[i]);
  return r;
}
```

## Events using delegates?

- Event systems are built on the notion of notification (call-back)
- Described with the **Observer** or **Publish/Subscribe** design pattern, as seen for JavaBeans
- A method invocation can be seen as a notification
- Event model introduced by Java 1.1:
  - There are source of events
  - There are listeners that ask sources for notifications
  - Event fires: a method is invoked for each subscriber

# Event Model


Subscriber
Subscribe
Event Source
Notification
Subscribed listeners

# Event model in Java

- Which method should call the event source to notify the event?
- In Java there are no delegates and interfaces are used instead (XXXListener)
- The listener must implement an interface and the source provides a method for (un)subscription.
- A list of subscribed listeners is kept by the event source

## Delegates to handle events

- Delegates allow connecting event sources to listeners independent of the types involved
- In C# a **delegate object** can be used to specify which method must be invoked when an event is fired
- One approach could be to store an array of delegates in the source to represents subscribers
- A component (not necessarily the listener) builds a delegate on the listener and subscribes to an event

## Multicast delegates

- Event notification is in general one-to-many
- CLR provides **multicast delegates** to support notification to many listeners
- A multicast delegate is a kind of delegate that holds inside a list of "delegate objects"
- Multicast delegates keep track of subscriptions to event sources reducing the burden of replicating the code

# Multicast delegates: Example

```
delegate void Event();
class EventSource {
   public Event evt;
   …
   evt(); // fires the event
   …
}

class Foo { public void MyMethod() {} }

// Elsewhere in the program!
EventSource src = new EventSource();
Foo f = new Foo();
src.evt += new Event(f.MyMethod); // subscribe
```

Unrelated types!

# C# and delegates

- In C# there is no way to choose between single and multicast delegates
- The compiler always generates multicast delegates
- If more than one method is registered, they are invoked on order of subscription. The returned value is the result of the last invokation.
- In principle JIT could get rid of possible inefficiencies

# Event keyword

- C# introduces the **event** keyword to control access to a delegate member.
- If a delegate field of a class is labeled with event then outside code will be able to use only += and -= operators on it
- Listener would not be allowed to affect the subscribers list in other ways
- Event infrastructures can be easily implemented by means of this keyword and delegates

## Multicast delegates: event keyword

```
delegate void Event();
class EventSource {
 public event Event evt;
 …
  evt(); // fires the event
 …
}

class Foo { public void MyMethod() {} }

// Elsewhere in the program!
EventSource src = new EventSource();
Foo f = new Foo();
src.evt += new Event(f.MyMethod);
src.evt = null; // ERROR!
```

# Remoting Connectors for .NET Distributed Components

- A **Remoting channel connection** allows a component or a client to access a remote component running in a different application domain in same or different processes.
- The marshaling makes it possible to invoke a remote method of a distributed component.
- There are two ways to marshal an object: in **MBV (Marshal by Value)** server passes a copy of object to client or in **Marshal by Reference (MBR)** client creates a proxy of a remote object.
- When a remote component must run at a remote site, MBR is the only choice.
- Similar to **RMI in Java**

# Remoting Asynchronous Callback Invocation Between Distributed .NET Components

- .NET supports remoting asynchronous callback, based on **Remoting Delegate**. It will not block out the client while waiting for notification from remote components.
- When client makes a synchronous call to remote method of remote component, it passes a **callback method** to the server to be called back later **through Remoting Delegate**.
- Typical scenarios:
  - The job on server will take very long to complete: remoting asynchronous callbacks allows the server to notify the client when the job is done.
  - The client wants to be notified once the stock prices reaches a specified level. Instead of pooling the stock price all the time, this technique allows the server to notify the client when this happens.

## Conclusion

- The Microsoft approach to components
- Several technologies developed around COM
- Main innovation: The .NET framework in 2000
- Common Language Runtime
- .NET components: Assemblies
- Composition by aggregation and containment
- Communication by Events and Delegates