.NET Framework Introduction

Gabriele Zannoni
gzannoni@deis.unibo.it

Premessa (nella vita, non si sa mai...)

- Non lavoro per Microsoft
- Non sono pagato da Microsoft
- Non sono un fan di Microsoft... ma mi piace lavorare con il .Net Framework

- Mi assumo la responsabilità di ciò che dico ma...

  "ambasciator non porta pena"!!!

Un po' di storia...

- .Net nasce intorno al 2000 per semplificare la vita ai programmatori Windows
- Prima del 2000 le alternative erano:
  - Sviluppo Windows:
    - C/C++ con Win32 (Windows API)
    - C++ con MFC (Microsoft Foundation Classes)
    - Visual Basic (e VBA)
    - Borland Delphi (l'unica vera alternativa)
    - Java?
  - Sviluppo Web:
    - ASP 3.0 con VSScript o JScript…
- Microsoft ha visto in Java un ottimo linguaggio, ha tentato di "portarlo" dentro le proprie tecnologie estendendolo al proprio bisogno…
- SUN non si è trovata molto d'accordo…

...un altro po' di storia...

- Necessità di avere un modello di sviluppo unificato – forms apps (Win32 – MFC), web (ASP 3.0), components (COM, COM+)
- Iniettare nuova linfa alle comunità di sviluppo – chi ha mai osato sviluppare codice più o meno open source per Win32 o MFC?

→ .Net Framework

Risultato

- Nascita di numerose comunità di sviluppatori .Net oriented
  - gotdotnet.com
  - ugidotnet.it
  - dotnetjunkies.com
  - ...
- Marcato incremento del codice open source sviluppato... anche da Microsoft
  - Application Blocks (Enterprise Library)
  - Starter Kits
  - Progetti SourceForge
  - ...
- Nascita di progetti indipendenti di porting della piattaforma .Net

General ideas and principles of .NET

- A compiler from any .Net language (C#, VB.Net) compiles into a common language - *CIL (Common Intermediate Language* or *MS IL)*
  - *MS IL ≈ Java Bytecode*
- Besides the CIL code, the compiler also generates *metadata* – information on types and named entities (classes, methods, fields, etc.)
- CIL + metadata + manifest = PE file (*Portable Executable)*
  - PE file ≈ Java JAR
- At runtime, the CIL code is *dynamically compiled (on-the-fly)* into a concrete target platform's native code (x86, Macintosh, IA-64, etc.) by a *Just-in-Time (JIT)* compiler
  - Java Bytecode is *normally* interpreted

Some advantages of .NET approach

- Metadata enable *managed-code (mode) execution*, with *runtime type-checking, runtime security checking, etc*.
- A software developer can develop modules in any language implemented for .NET and include into applications any modules developed in other languages also implemented for .NET

The Intermediate Language Magic!

Basic notions and acronyms used for .NET

- CLI (Common Language Infrastructure) – an ECMA standard to implement the .NET common infrastructure for all languages (ECMA-335)
- CTS (Common Type System) – the common type system for .NET (ISO/ECMA standard)
- CLR (Common Language Runtime) – the common runtime environment for .NET (ISO/ECMA standard):
  - cross-language integration
  - code access security
  - object lifetime management (GC)
  - resouce management
  - type safety
  - pre-emptive threading
  - metadata services (type reflection)
  - debugging and profiling support
- CIL (MS IL) – the common intermediate language for .NET based on postfix (reverse Polish) notation

Standard ECMA-335

- CLI is a **runtime environment**, with:
  - a file format
  - a common type system (CTS)
  - an extensible metadata system
  - an intermediate language (CIL)
  - access to the underlying platform
  - a factored base class library (BCL – Base Class Library)

Basic notions and acronyms used for .NET

- Metadata – a common (table-based) representation of information on the types and the named entities defined and used in an application
- Manifest – contents (inventory) of an assembly, in particular, of a PE file representing an assembly
- PE (Portable Executable) – the file to contain CIL code, metadata, resources and manifest
- Assembly – a representation of an application in a form of a PE file or a set of PE files, generated by a .NET compiler

“Comparison” of .NET and Java 2 platforms

- .NET – *multi-language* programming
- Java 2 – programming *in Java only* (except for JNI)

- .NET – *a universal type system and universal data representation*
- in Java 2 everything is *Java-oriented only*

- Java and C# languages: C# has taken everything useful from Java, but it contains additional features (properties, indexers, attributes aliases, extra kinds of statements, etc.)

Indipendenza dalla piattaforma?

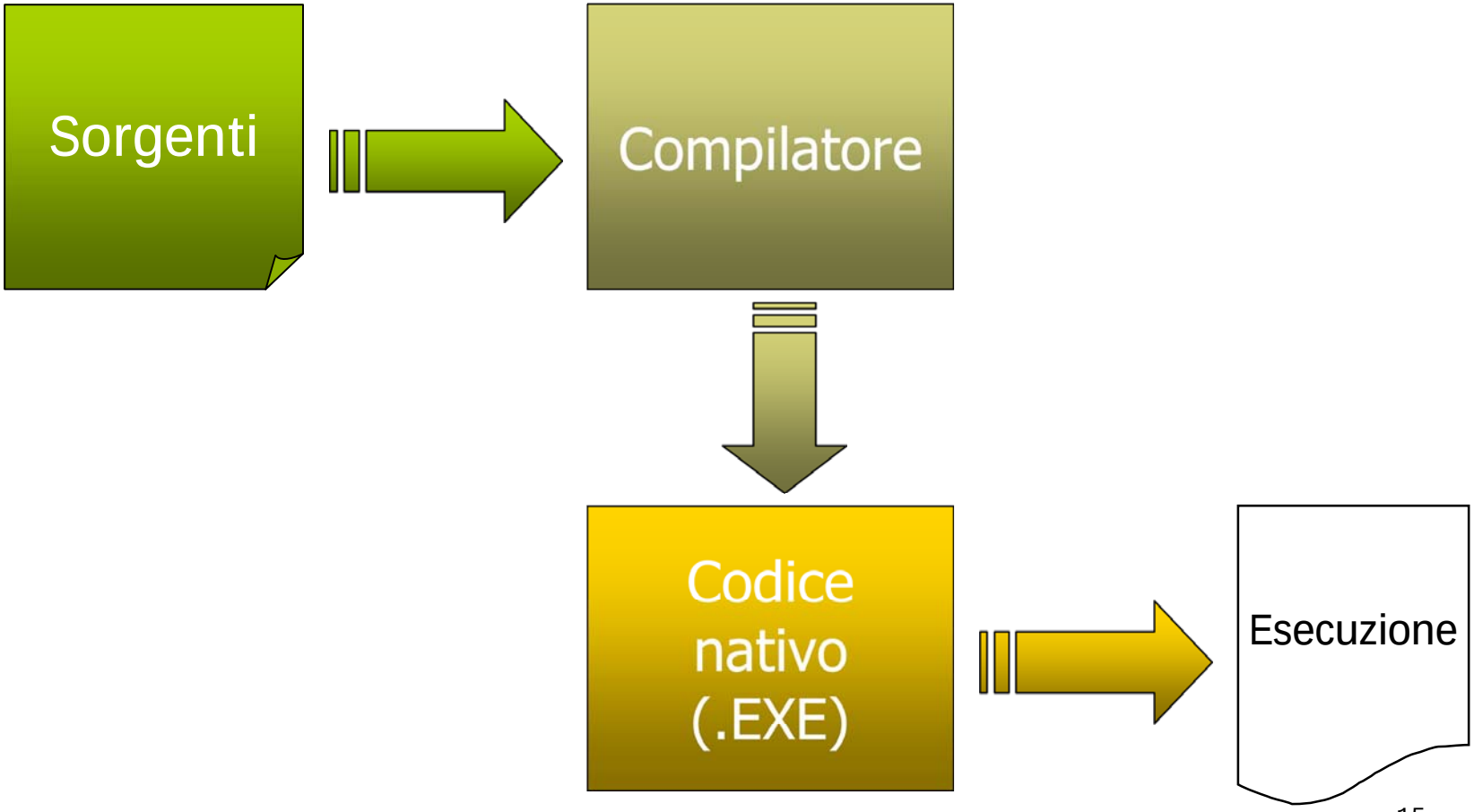
- .NET è una possibile implementazione di CLI (*Common Language Infrastructure)*
- Anche il linguaggio C# è uno standard ECMA (ECMA-334)
- Esistono altre implementazioni di CLI:
  - **SSCLI** (Shared Source CLI by Microsoft, per Windows, FreeBSD e Macintosh) - **Rotor**
  - **Mono** (per Linux)
  - **DotGNU**
  - **Intel OCL** (Open CLI Library)
  - ...

Piattaforma multi-linguaggio

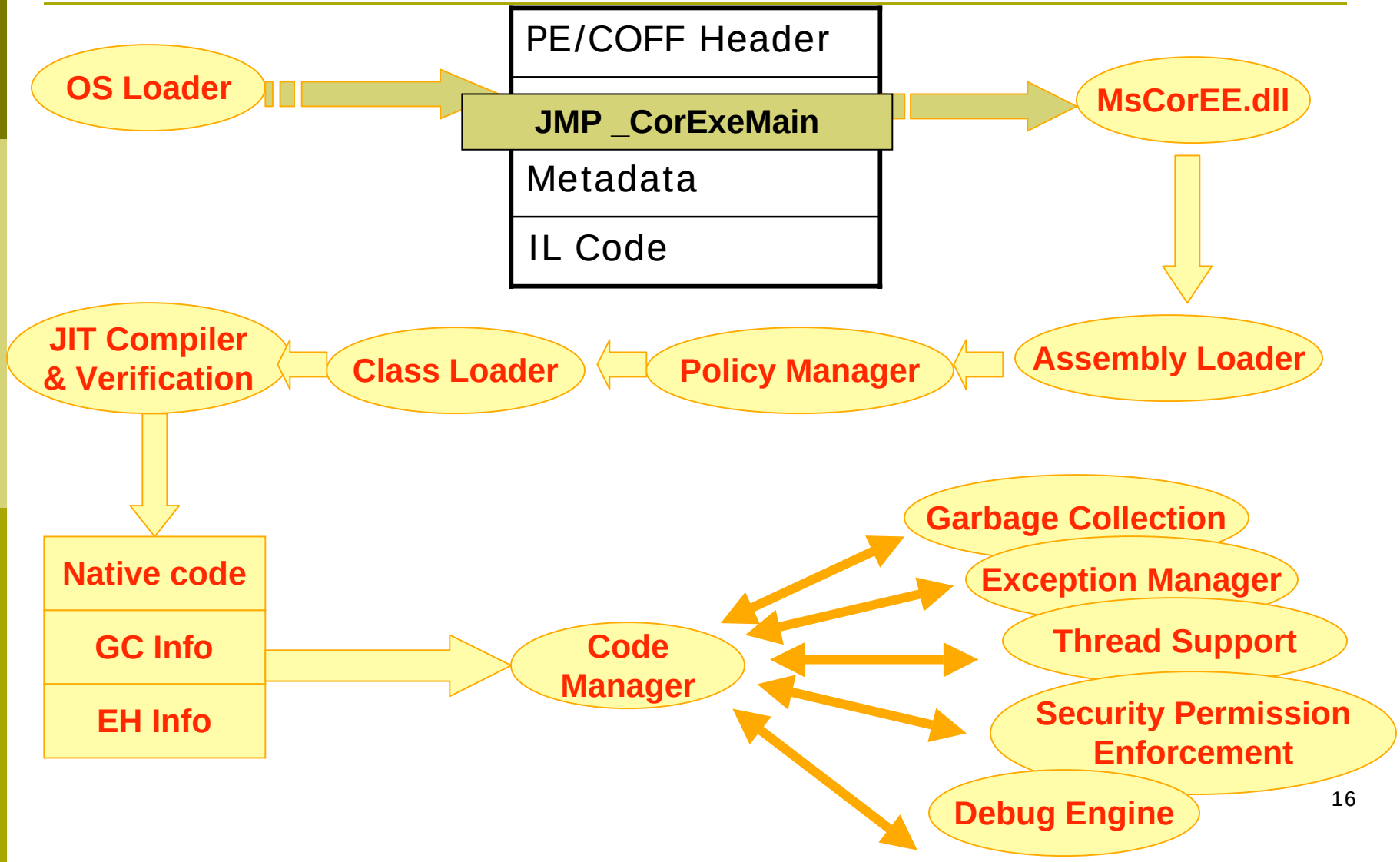
- Libertà di scelta del linguaggio
  - Non tutte le funzionalità del *framework* .NET sono disponibili a tutti i linguaggi .NET
  - I componenti di un'applicazione possono essere scritti con diversi linguaggi
  - Interoperabilità garantita se linguaggi usati sono almeno CLS compliant (Common Language Specification)
- Impatto sui tool
  - Tool disponibili per tutti i linguaggi: Debugger, Profiler, ecc.

Codice interpretato

Codice nativo



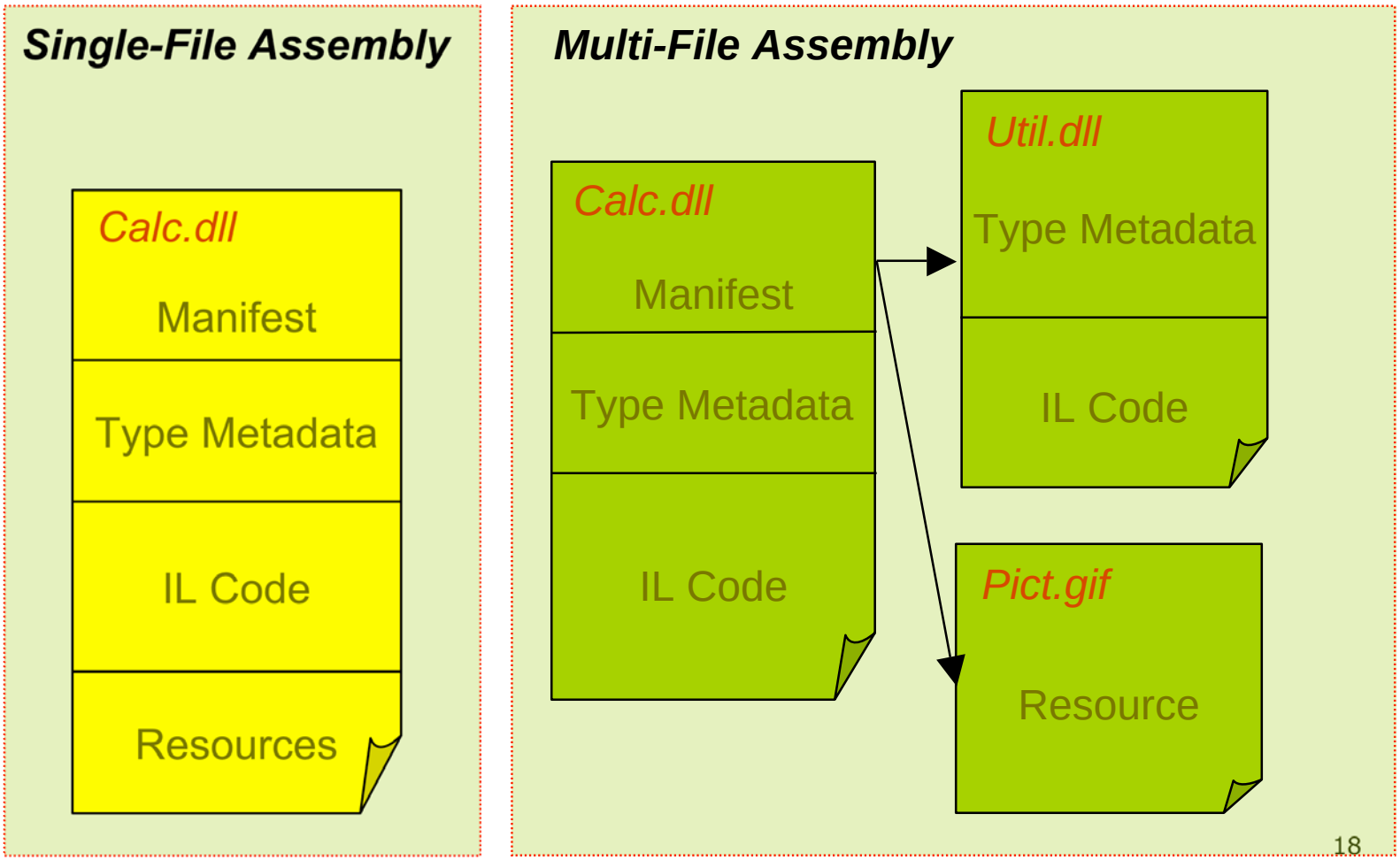
Managed Code: RUN!



Assembly

- Unità minima per la distribuzione, il versioning e la security
  - 1+ file
- ***Manifest***
  - Metadati che descrivono l'assembly stesso
- ***Type metadata***
  - Metadati che descrivono completamente tutti i tipi contenuti nell'assembly
- **Codice in *Intermediate Language***
  - Ottenuto da un qualsiasi linguaggio di programmazione
- **Risorse**
  - Immagini, icone, ...

Assembly


Single-File Assembly
Calc.dll
Manifest
Type Metadata
IL Code
Resources
Multi-File Assembly
Calc.dll
Manifest
Type Metadata
IL Code
Util.dll
Type Metadata
IL Code
Pict.gif
Resource

Assembly

```
.assembly Hello { }
.assembly extern mscorlib { }
.method public static void main()
{
  .entrypoint
  ldstr "Hello IL World!"
  call void [mscorlib]System.Console::WriteLine
  (class System.String)
  ret
}


ilasm helloil.il
```

Assembly

- **Assembly privati**
  - Utilizzati da un'applicazione specifica
  - *Directory* applicazione (e *sub-directory*)
- **Assembly condivisi**
  - Utilizzati da più applicazioni
  - ***Global Assembly Cache*** (**GAC**)
  - c:\windows\assembly
- **Assembly scaricati da URL**
  - *Download cache*
  - c:\windows\assembly\download
- GACUTIL.EXE
  - *Tool* a linea di comando per esaminare GAC e *download cache*

Deployment semplificato

- Installazione senza effetti collaterali
  - Applicazioni e componenti possono essere
    - condivisi
    - privati
- Esecuzione *side-by-side*
  - Diverse versioni dello stesso componente possono essere usate anche dallo stesso assembly

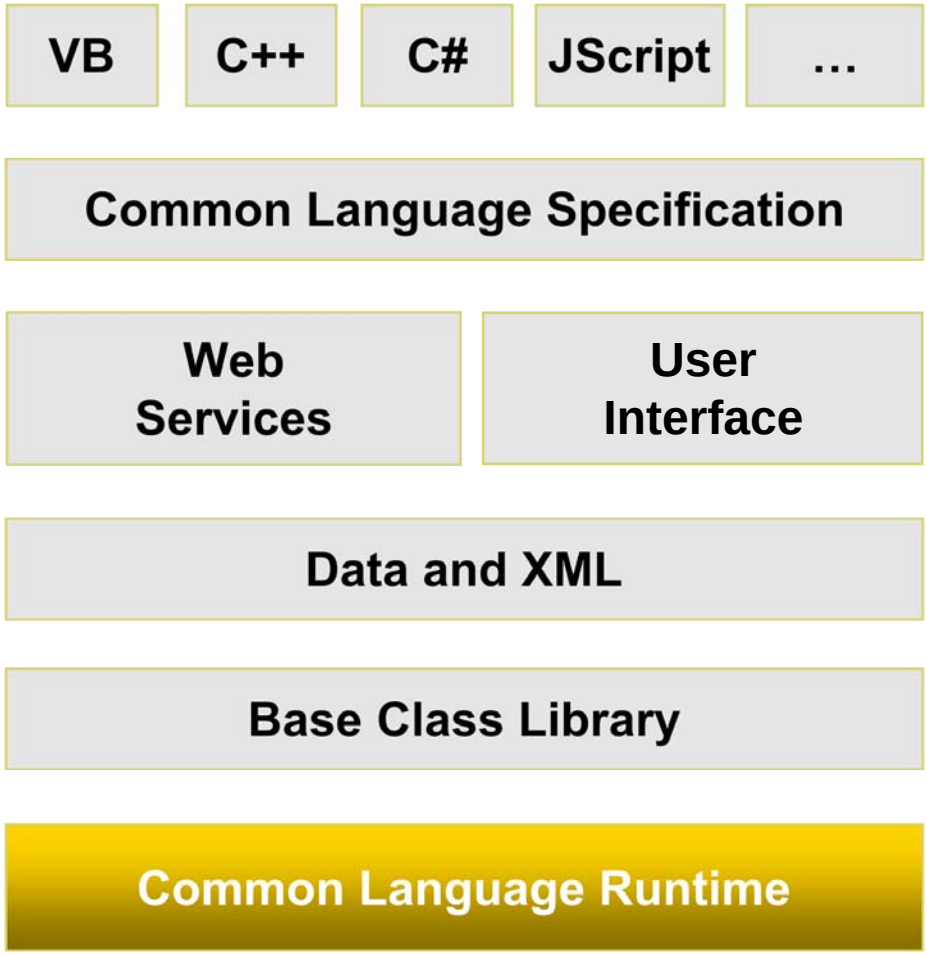
Metadati

- Descrizione dell'assembly - Manifest
  - Identità: nome, versione, cultura [, public key]
  - Lista dei file che compongono l'assembly
  - Riferimenti ad altri assembly da cui si dipende
  - Permessi necessari per l'esecuzione
- Descrizione dei tipi contenuti nell'assembly
  - Nome, visibilità, classe base, interfacce
  - Campi, metodi, proprietà, eventi, …
- Attributi
  - Definiti dal compilatore
  - Definiti dal framework
  - Definiti dall'utente

Tool che usano i metadati

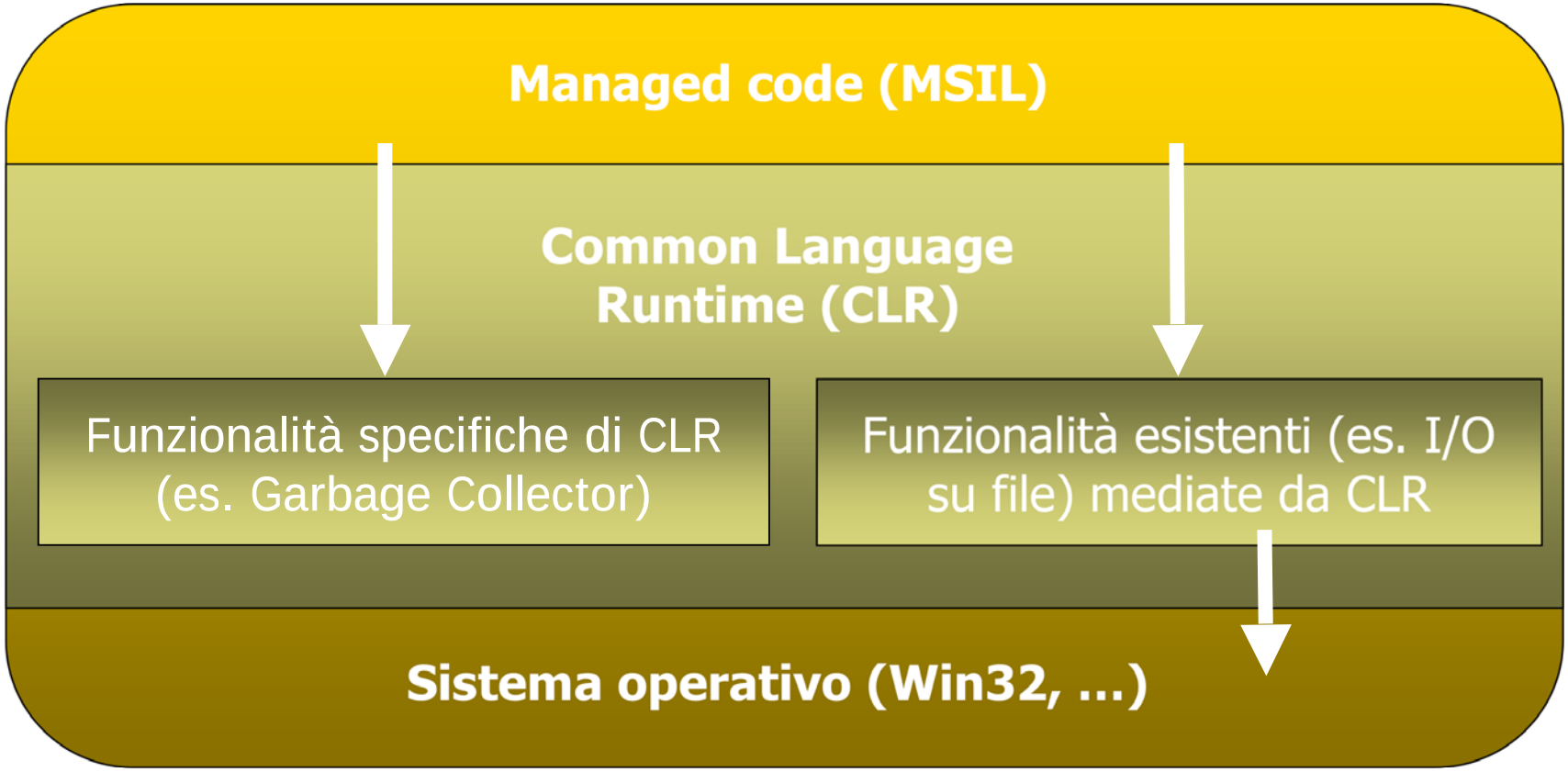
- Compilatori
  - Compilazione condizionale
- Ambienti RAD
  - Informazioni sulle proprietà dei componenti
    - Categoria
    - Descrizione
  - Editor personalizzati di tipi di proprietà
- Analisi dei tipi e del codice
  - Intellisense
  - ILDASM
  - Anakrino, Reflector

Common Language Runtime

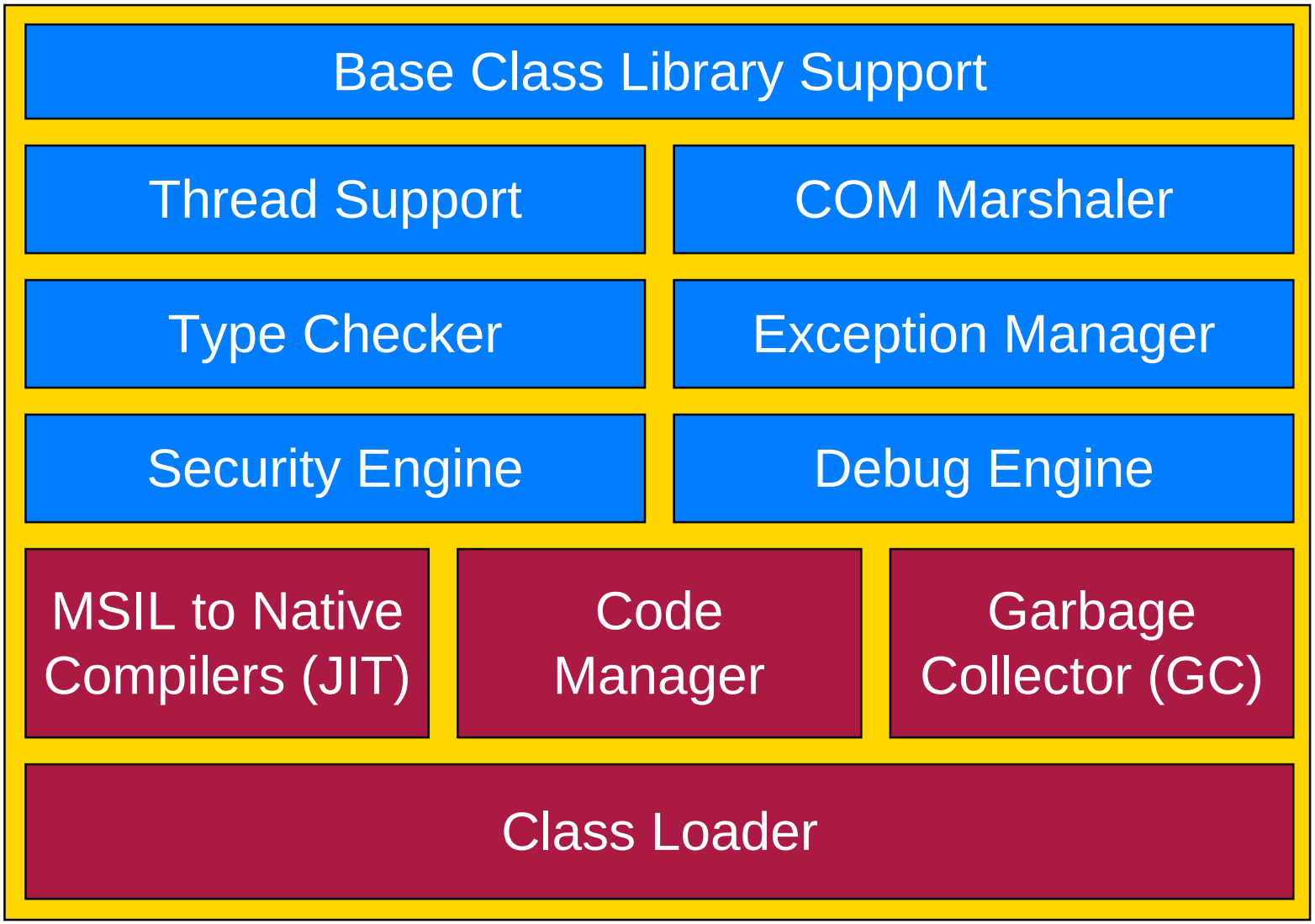


Common Language Runtime

- IL CLR offre vari servizi alle applicazioni



Common Language Runtime



Sicurezza e affidabilità del codice

- Separazione spazi di memoria in un processo con **`AppDomain`**
- Controllo del codice e controllo dei tipi
  - *Cast* non sicuri
  - Variabili non inizializzate
  - Accessi ad *array* oltre i limiti di allocazione
- Code Access Security
  - Verifica automatica che il codice possa essere eseguito in sicurezza dall'utente

Garbage Collector

- *Garbage Collector* per tutti gli oggetti .NET
- Gestione del ciclo di vita degli oggetti
- Gli oggetti vengono distrutti automaticamente quando non sono più referenziati
- Algoritmo *Mark-and-Compact*

Garbage Collector e distruzione deterministica

- In alcuni casi serve un comportamento di finalizzazione deterministica
  - Riferimenti a oggetti non gestiti
  - Utilizzo di risorse che devono essere rilasciate appena termina il loro utilizzo
- Non è possibile utilizzare il metodo **`Finalize`**

  (in C# il distruttore), in quanto non è richiamabile direttamente
- È necessario implementare l'interfaccia **`IDisposable`**

Gestione delle eccezioni

- Eccezione è una classe derivata da System.Exception
- Concetti universali
  - Lanciare un'eccezione (**`throw`**)
  - Catturare un'eccezione (**`catch`**)
  - Eseguire codice di uscita da un blocco controllato (**`finally`**)
- Disponibile in tutti i linguaggi .NET con sintassi diverse

Altri servizi del CLR

- ***Reflection***
  - Analisi dei metadati di un assembly
  - Generazione di un assembly dinamico
- ***Remoting***
  - Chiamata di componenti remoti (.NET)
- **Interoperabilità** (COM, *Platform Invoke*)

Reflection

- È possibile interrogare un assembly caricato in memoria
  - Tipi (classi, interfacce, enumeratori, etc.)
  - Membri (attributi, proprietà, metodi, etc.)
  - Parametri
- È possibile forzare il caricamento in memoria di un assembly con i metodi **`Load`**/**`LoadFrom`**

Common Type System

- Tipi di dato supportati dal *framework* .NET
  - Alla base di tutti i linguaggi .NET
- Consente di fornire un modello di programmazione unificato
- Progettato per linguaggi object-oriented, procedurali e funzionali
  - Esaminate caratteristiche di 20 linguaggi
  - Tutte le funzionalità disponibili con IL
  - Ogni linguaggio utilizza alcune caratteristiche
- Common Language Specification
  - Regole di compatibilità tra linguaggi
  - Sottoinsieme di CTS

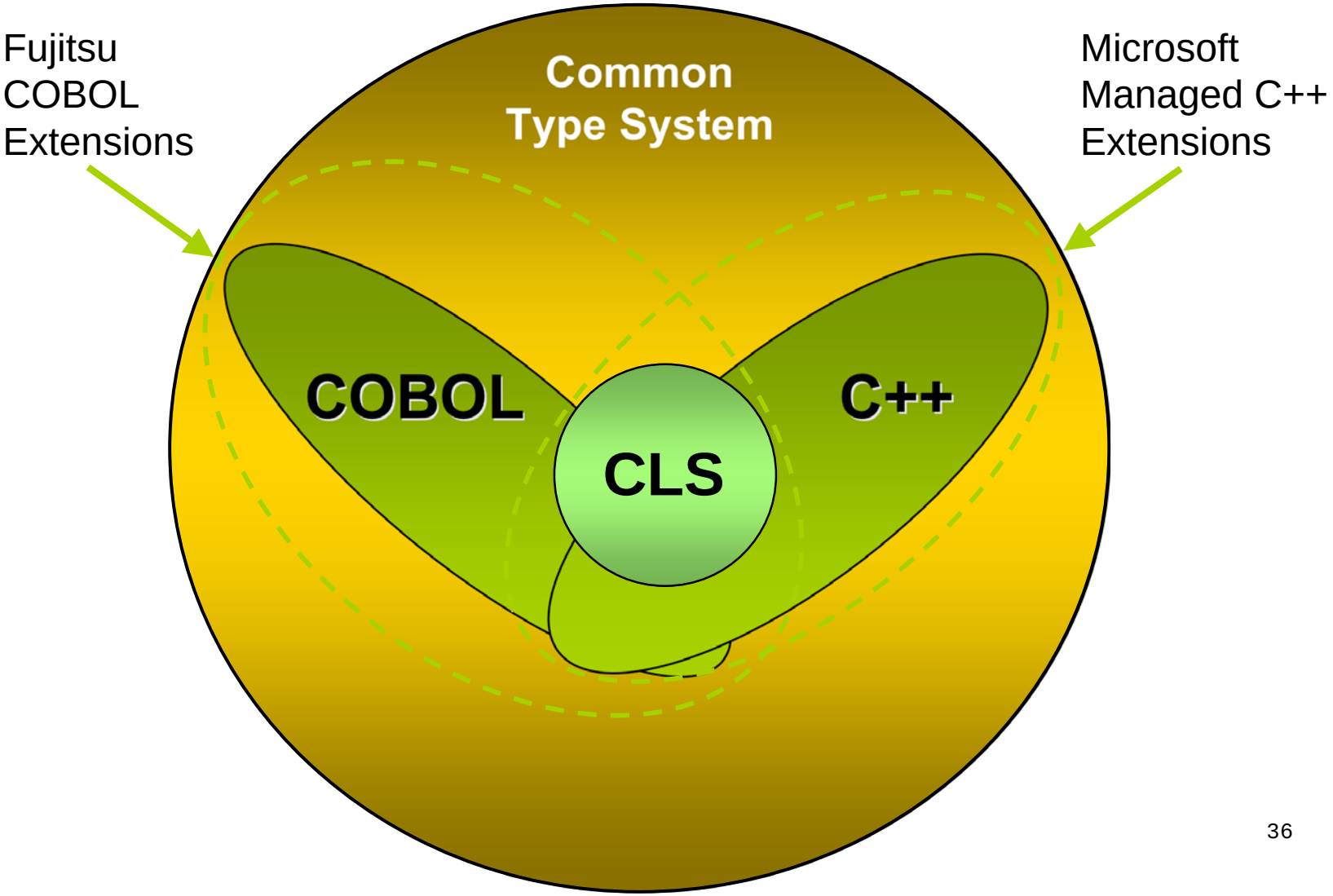
Common Type System

- Alla base di tutto ci sono i tipi: **classi**, **strutture**, **interfacce**, **enumerativi**, **delegati**
- Fortemente tipizzato (*compile-time*)
- *Object-oriented*
  - Campi, metodi, tipi nidificati, proprietà, ...
- *Overload* di funzioni (*compile-time*)
- Invocazione metodi virtuali risolta a *run-time*
- Ereditarietà singola
- Ereditarietà multipla di interfacce
- Gestione strutturata delle eccezioni

Common Language Specification

- Regole per gli identificatori
  - Unicode, *case-sensitivity*
  - *Keyword*
- Regole di *overload* più restrittive
- Nessun puntatore *unmanaged*
- Devono essere supportate interfacce multiple con metodi dello stesso nome
- Regole per costruttori degli oggetti
- Regole per denominazione proprietà ed eventi

Common Language Specification



Common Type System
Tipi nativi

CTS	C#
System.Object	object
System.String	string
System.Boolean	bool
System.Char	char
System.Single	float
System.Double	double
System.Decimal	decimal
System.SByte	sbyte
System.Byte	byte
System.Int16	short
System.UInt16	ushort
System.Int32	int
System.UInt32	uint
System.Int64	long
System.UInt64	ulong

Common Type System

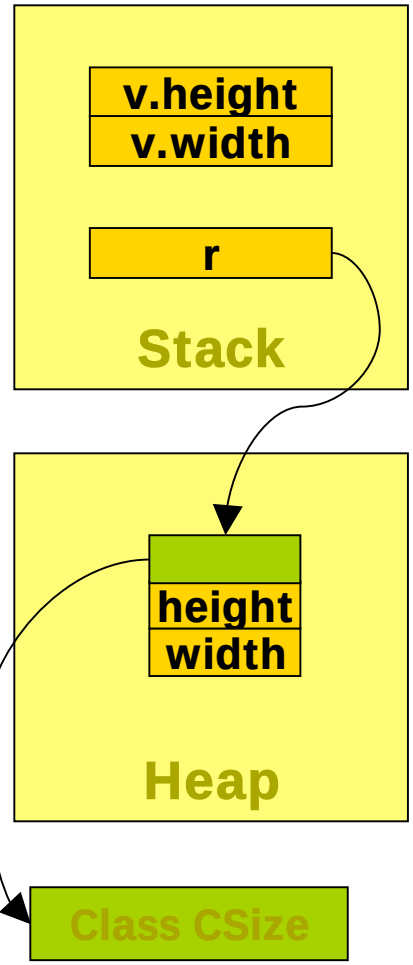
- Tutto è un oggetto
  - **`System.Object`** è la classe radice
- Due categorie di tipi
  - **Tipi riferimento**
    - Riferimenti a oggetti allocati sull'***heap*** gestito
    - Indirizzi di memoria
  - **Tipi valore**
    - Allocati sullo ***stack*** o parte di altri oggetti
    - Sequenza di byte

Common Type System
Tipi valore

- I tipi valore comprendono:
  - Tipi primitivi (*built-in*)
    - Int32, …
    - Single, Double
    - Decimal
    - Boolean
    - Char
  - Tipi definiti dall'utente
    - Strutture (**`struct`**)
    - Enumerativi (**`enum`**)

Common Type System
Tipi valore vs tipi riferimento

```
public struct Size
{
  public int height;
  public int width;
}
public class CSize
{
  public int height;
  public int width;
}
public static void Main()
{
  Size v;            // v istanza di Size
  v.height = 100;    // ok
  CSize r;           // r è un reference
  r.height = 100;    // NO, r non assegnato
  r = new CSize();   // r fa riferimento a un CSize
  r.height = 100;    // ok, r inizializzata
}
```

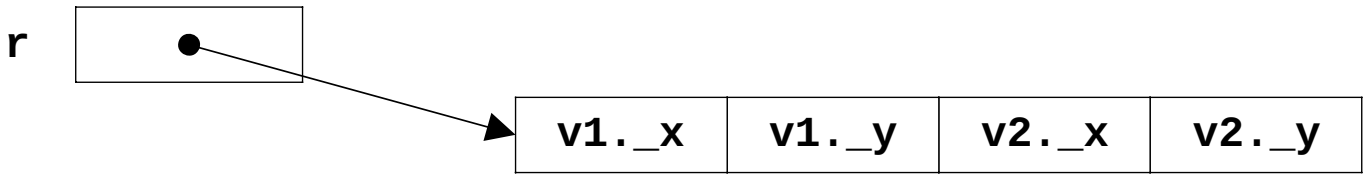


Common Type System
Tipi valore vs tipi riferimento

```
public struct Point
{
  private int _x, _y;

  public Point(int x, int y)
  {
    _x = x;
    _y = y;
  }

  public int X
  {
    get { return _x; }
    set { _x = value; }
  }

  public int Y
  {
    get { return _y; }
    set { _y = value; }
  }
}
```

Common Type System
Tipi valore vs tipi riferimento

```
public class Rectangle
  {
  Point v1;
  Point v2;
  …
  }
…
Rectangle r = new Rectangle();
```



Common Type System
Tipi valore vs tipi riferimento

```
…
Point[] points = new Point[100];
for (int i = 0; i < 100; i++)
  points[i] = new Point(i, i);
…
```

- Alla fine, rimane 1 solo oggetto nell'*heap* (l'array di **Point**)

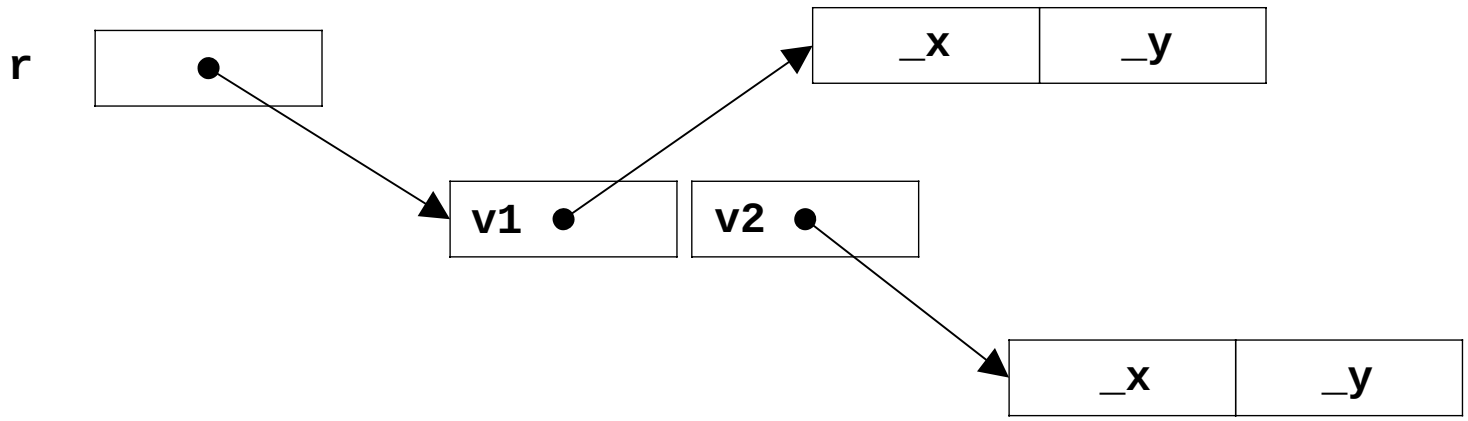
```
…
Point[] points = new Point[100];
for (int i = 0; i < 100; i++)
  {
  points[i].X = i;
  points[i].Y = i;
  }
…
```

Common Type System
Tipi valore vs tipi riferimento

```
public class Point
{
  private int _x, _y;

  public Point(int x, int y)
  {
    _x = x;
    _y = y;
  }

  public int X
  {
    get { return _x; }
    set { _x = value; }
  }

  public int Y
  {
    get { return _y; }
    set { _y = value; }
  }
}
```

Common Type System
Tipi valore vs tipi riferimento

```
public class Rectangle
  {
  Point v1;
  Point v2;
  …
  }
…
Rectangle r = new Rectangle();
```



Common Type System
Tipi valore vs tipi riferimento

```
…
Point[] points = new Point[100];
for (int i = 0; i < 100; i++)
  points[i] = new Point(i, i);
…
```

- Alla fine, rimangono 101 oggetti nell'*heap* (1 array di **Point** + 100 **Point**)

```
…
Point[] points = new Point[100];
for (int i = 0; i < 100; i++)
  {
  points[i].X = i;
  points[i].Y = i;
  }
…
```

NO!

Boxing / Unboxing

- Un qualsiasi tipo valore può essere automaticamente convertito in un tipo riferimento (***boxing***) mediante un *up cast* implicito a **`System.Object`**

```
int i = 123;
object o = i;
```

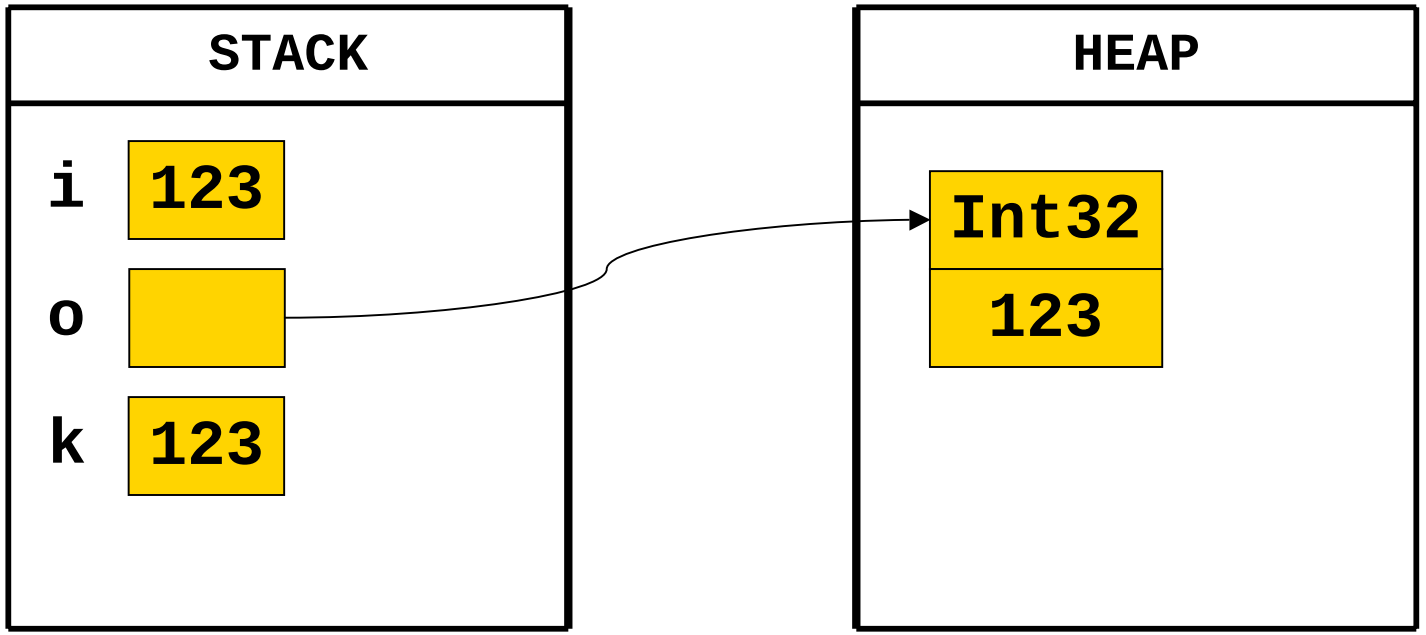
- Un tipo valore "*boxed*" può tornare ad essere un tipo valore standard (***unboxing***) mediante un *down cast* esplicito

```
int k = (int) o;
```

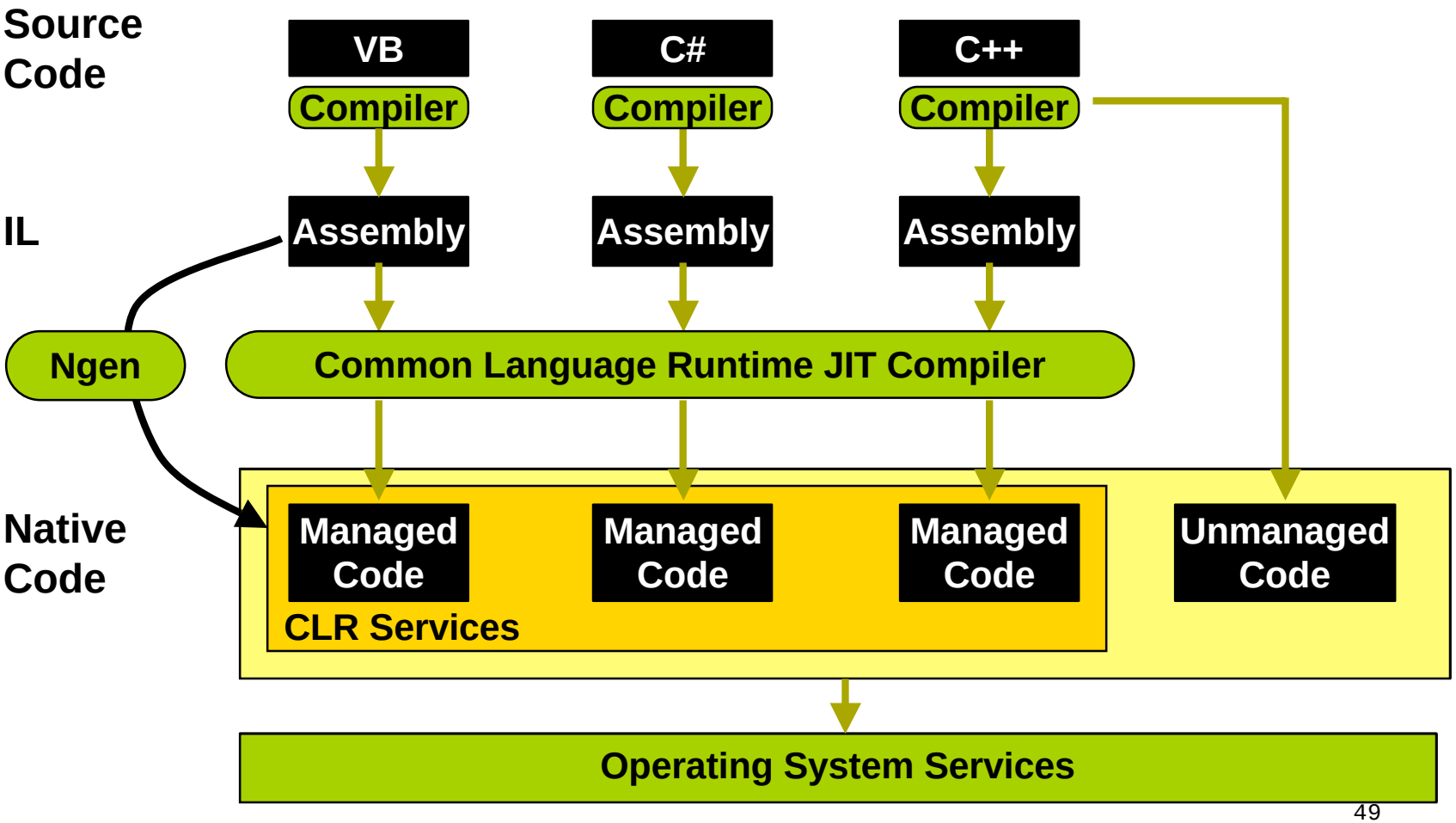
- Un tipo valore "*boxed*" è un **clone indipendente**

Boxing / Unboxing

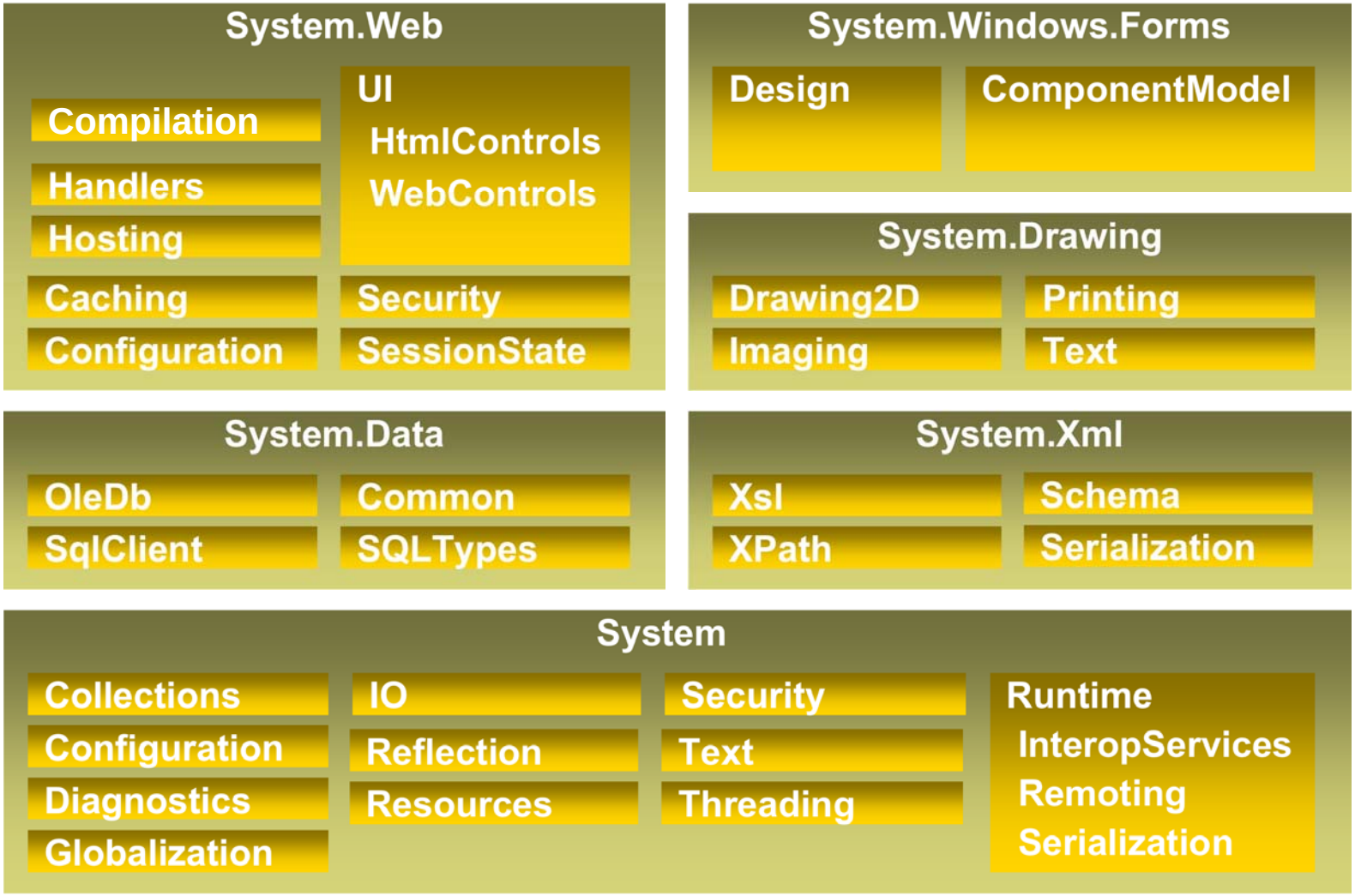
```
int i = 123;
object o = i;
int k = (int) o;
```



Modello di esecuzione



Framework .NET



C# VS Java

- Proprietà
- Indexers
- Possibilità di static binding (il default)
- Implementazione interfacce in modo implicito o esplicito
- Delegati & Eventi (l'observer è gratis!)
- Namespaces & Assemblies
- Distruzione deterministica (quasi...)

Indexers

```
class A {
  string this[int i, string s] {
     get { ... return ... }
     set { ... = value; ... }
  }
}
...
A anInstance = new A();
string s = anInstance[10, “Java”];
```

Static & Dynamic Binding

```
class A {
   public void Method1() {...}
   public virtual void Method2() {...}
}
class B : A {
   public new void Method1() {...}
   public override void Method2() {...}
}
class C : B {
   public new void Method2() {...}
}
class D : C {
   public new virtual void Method1() {...} //??
   public new virtual void Method2() {...} //??
}
class E : D {
   public override void Method1() {...} //??
   public override void Method2() {...} //??
}
```

Interfacce

- Possono contenere eventi, proprietà, metodi, indexers

```
interface I1 {
   void Method();
}
interface I2 {
   void Method();
}
class A : I1, I2 {
   public void Method() {...} //implicit
   void I2.Method() {...}     //explicit
}
```

Delegates and events

- *History*: Java – events handled by event listeners that subscribe the event.
  - Mechanism not part of the Java language semantics

- Delegates in .NET CTS (multi-language concept) → delegates are methods to whom event handling is delegated
  - Similar to function pointers in C and C++
    - Their use is type safe because they are *managed pointers*: *(ref to metadata to the method type, ref to method)*
  - Multicast delegates call more than one method
    - Subclasses of *System.MulticastDelegate*
    - Cause execution of the *Combine* method from *MulticastDelegate*

- Events
  - Change the state of the object
  - Observer registers the delegate and the event
    - Event generation includes MulticastDelegate

Delegates

- A delegate variable can have the value *null* (no method assigned).
- If *null*, a delegate variable must not be called (otherwise exception).
- Delegate variables are first class objects: can be stored in a data structure, passed as a parameter, etc.

```
delegate void MyDelegate(int i, string s);
...
void Method(int i, string s) {...}
...
MyDelegate myd = new MyDelegate(Method);
myd(10, “Ziao!”); //Puntatore a funzione?
```

Delegates

```
m = new DelegateType (obj.Method);// or just m = obj.Method;
```

- A delegate variable stores a method and its receiver, but no parameters !
  new Notifier(myObj.SayHello);
- *obj* can be *this* (and can be omitted)
  new Notifier(SayHello);
- *Method* can be *static*. In this case the class name must be specified instead of *obj*.
  new Notifier(MyClass.StaticSayHello);
- *Method* must not be *abstract*, but it can be *virtual*, *override*, or *new*.
- *Method* signature must match the signature of the *DelegateType*
  - same number of parameters
  - same parameter types (including the return type)
  - same parameter kinds (value, ref/out)

Eventi

- Field speciali di tipo delegato

```
class A {
   public event MyDelegate MyEvent;
   public void Fire() { OnMyEvent(19, “Zut!”); }
   protected void OnMyEvent(int i, string s){
       if (MyEvent != null)
             MyEvent(i, s);
   }
}
...
A anInstance = new A();
anInstance.MyEvent += new MyDelegate(Method);
anInstance.Fire();
anInstance.MyEvent -= new MyDelegate(Method);
```

C# Namespaces VS Java Packages

C#

A file may contain multiple namespaces

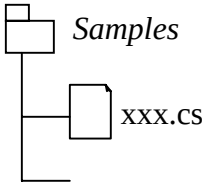
xxx.cs

```
namespace A {...}
namespace B {...}
namespace C {...}
```

Namespaces and classes are not mapped to directories and files

xxx.cs

```
namespace A {
    class C {...}
}
```



Java

A file may contain just 1 package

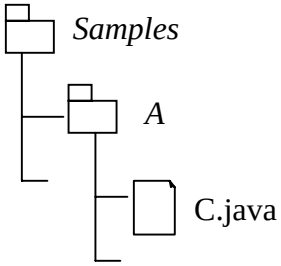
xxx.java

```
package A;
...
...
```

Packages and classes are mapped to directories and files

C.java

```
package A;
class C {...}
```



Namespaces vs. Packages (continued)

C#

Imports *namespaces*

```
using System;
```

Namespaces are imported in other Namesp.

```
namespace A {
    using C; // imports C into A
}            // only in this file
namespace B {
    using D;
}
```

Alias names allowed

```
using F = System.Windows.Forms;
...
F.Button b;
```

for explicit qualification and short names

Java

Imports *classes*

```
import java.util.LinkedList;
import java.awt.*;
```

Classes are imported in files

```
import java.util.LinkedList;
```

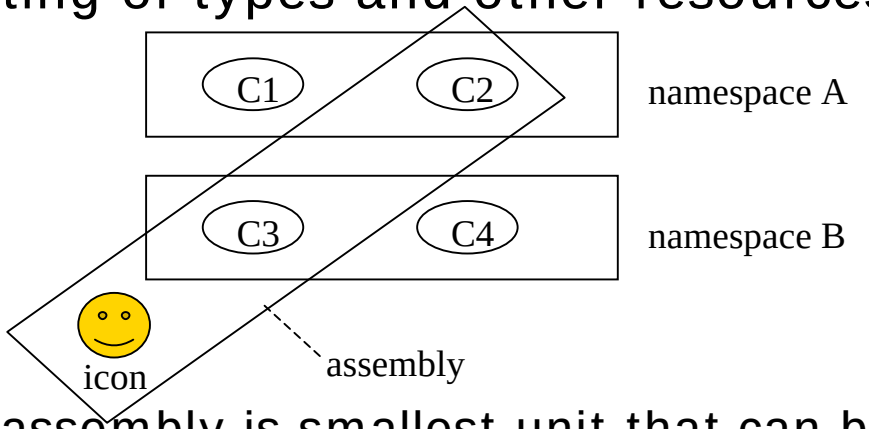
Java has visibility *package*

```
package A;
class C {
    void f() {...}  // package
}
```

C# has only visibility *internal* (!= namespace)

Assemblies

Run time unit consisting of types and other resources (e.g. icons)



An assembly is a

- Unit of deployment: assembly is smallest unit that can be deployed individually
- Unit of versioning: all types in an assembly have the same version number

  - one assembly may contain multiple namespaces
  - one namespace may be spread over several assemblies
  - an assembly may consist of multiple files, held together by a *manifest* ("table of contents")

Assembly ≈ JAR file in Java
Assembly ≈ Component in .NET

Attributes

User-defined metainformation about program elements

- Can be attached to types, members, assemblies, etc.
- Can be queried at run time (reflection)
- Are implemented as classes that are derived from *System.Attribute*.
- Are stored in the metadata of an assembly.
- Often used by CLR services (serialization, remoting, COM interoperability)

Example

```
[Serializable]
class C {...}                    // makes the class serializable
```

Also possible to attach multiple attributes

```
[Serializable] [Obsolete]
class C {...}

[Serializable, Obsolete]
class C {...}
```

Defining Your Own Attributes

Declaration

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Interface, Inherited=true)]
class Comment : Attribute {
    string text, author;
    public string Text { get {return text;} }
    public string Author { get {return author;} set {author = value;} }
    public Comment (string text) { this.text = text; author ="HM"; }
}
```

Usage

```
[Comment("This is a demo class for Attributes", Author="XX")]
class C { ... }
```

Querying the attribute at runtime

```
class Test {

    static void Main() {
        Type t = typeof(C);
        object[] a = t.GetCustomAttributes(typeof(Comment), true);
        Comment ca = (Comment)a[0];
        Console.WriteLine(ca.Text + ", " + ca.Author);
    }
}
```

search should also be continued in subclasses