Introduzione al linguaggio C#

Anno Accademico 2017-18

Obiettivi

- Conoscere le caratteristiche base del linguaggio
  - Metodi, proprietà, eventi, attributi
  - Pattern di programmazione
- Comprendere l'utilizzo delle principali classi legate all'ambiente .NET
  - Oggetti, stringhe, file, interfacce grafiche
- Sapere realizzare semplici applicazioni dotate di interfaccia grafica

Un linguaggio “a componenti”…

- Supporto di astrazioni di livello elevato
  - Metodi, eventi, proprietà, campi
  - Attributi (metadati)
- Permette la realizzazione di pacchetti entro-contenuti ed autodescrittivi (assembly)
  - Non richiede l’uso di linguaggi ad hoc per la descrizione delle interfacce
  - Supporta la documentazione integrata con il codice

… con accesso uniforme ai dati…

- In Java poca uniformità tra i dati
  - I tipi elementari (int, byte, char, boolean,…) non sono oggetti
  - Esistono classi “wrapper” (Integer, Byte, …) con accesso in sola lettura:
    - Necessari per gestire collezioni (liste, vettori, …) di dati elementari
- In C#, le differenze sono meno nette
  - La conversione da valore elementare ad oggetto è trasparente (boxing, anche in Java dalla versione 1.5)
  - La conversione inversa (unboxing) è esplicita (richiede uso di casting, idem)
  - Le istanze delle classi wrapper (Int32, Byte,…) possono essere modificate
  - Le collezioni di dati sono più semplici da gestire

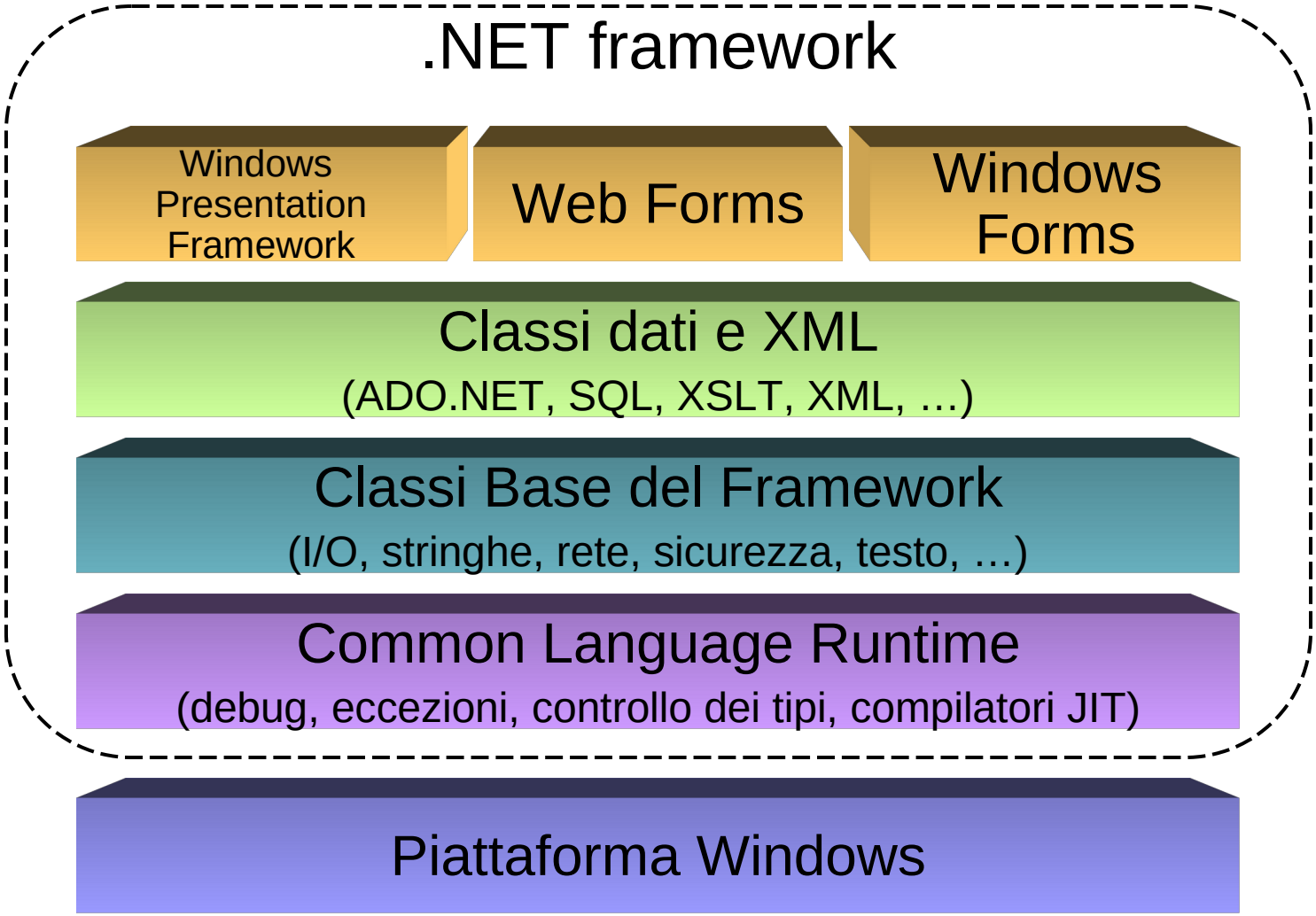
… robusto…

- Gestione automatica dell'allocazione di memoria
  - Non è una novità per i programmatori Java!
- Il compilatore verifica la corretta inizializzazione delle variabili
  - Idem
- Il concetto di eccezione è cablato nel linguaggio
  - Ma non è obbligatorio né dichiararne né gestire eventuali malfunzionamenti
- Ogni modulo binario ha esplicitamente una versione
  - È compito del programmatore gestire eventuali conflitti

…compatibile con gli investimenti precedenti

- Sintassi simile al C/C++/Java
  - Concetti simili, nomi differenti
  - Supporto della modalità non gestita: accesso ai puntatori
- Alto livello di interoperabilità
  - Con gli altri linguaggi .NET
  - Con altri standard (XML, COM, …)
- Curva di apprendimento rapida
  - .NET è costituito da milioni di righe di codice C#

Architettura .NET

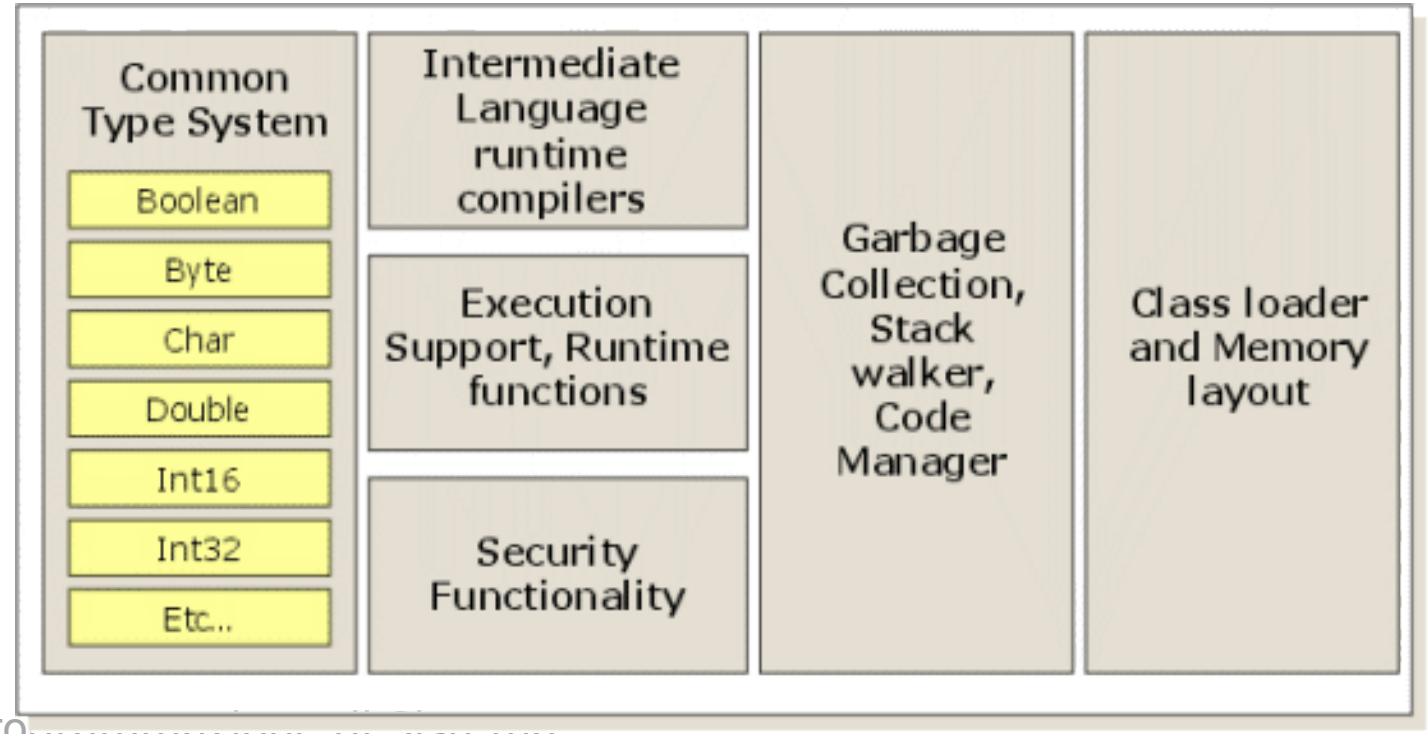


Alla base di C#: Common Language Runtime (1)

- Strato software che si interpone tra il sistema operativo e le applicazioni .NET
- La fase di compilazione genera dei moduli espressi in un linguaggio intermedio comune (CIL)
- Il CLR contiene un modulo che traduce il codice intermedio nel linguaggio macchina del processore che ospita l'applicazione, permettendo così l'esecuzione
- Esiste un sistema comune di tipi e di API utilizzabili dai linguaggi supportati dal framework

Alla base di C#: Common Language Runtime (2)

- Fornisce anche ulteriori strumenti per lo sviluppo, riducendo i problemi di installazione e compatibilità



Managed Code

- Il codice utilizzato per la costruzione di applicazioni .NET è detto “managed”
  - Gestione dell'esecuzione delle singole istruzioni virtuali
  - Gestione automatica del ciclo di vita degli oggetti e della relativa distruzione da parte del garbage collector
  - Gestione strutturata delle eccezioni

CIL

- Common Intermediate Language
  - Set di istruzioni di un elaboratore virtuale in cui vengono compilati tutti i sorgenti dei linguaggi .NET
  - Elimina le dipendenze dalla CPU
  - Equivalente del bytecode di Java
- Contiene le istruzioni per il caricamento, la memorizzazione, l'inizializzazione delle classi e le chiamate ai metodi
- Combinato con i metadati e il sistema di tipi comune, permette l'integrazione moduli scritti in linguaggi differenti
- Per poter eseguire una applicazione, il codice intermedio corrispondente è convertito in codice macchina (compilazione just in time)

C# in sintesi

- Linguaggio ad oggetti fortemente tipato...
  - L'unità minima di programmazione è la classe
- ...ad ereditarietà semplice...
  - Tutte le classi derivano da "System.Object"
- ...con supporto della riflessione...
  - È possibile esaminare ogni oggetto, scoprirne le caratteristiche, costruire nuove classi in fase di esecuzione
- ...che incorpora nella propria sintassi i principali pattern di programmazione
  - Eventi, eccezioni, iterazione, gestione della memoria e delle risorse, proprietà, metadati, ...

Un esempio

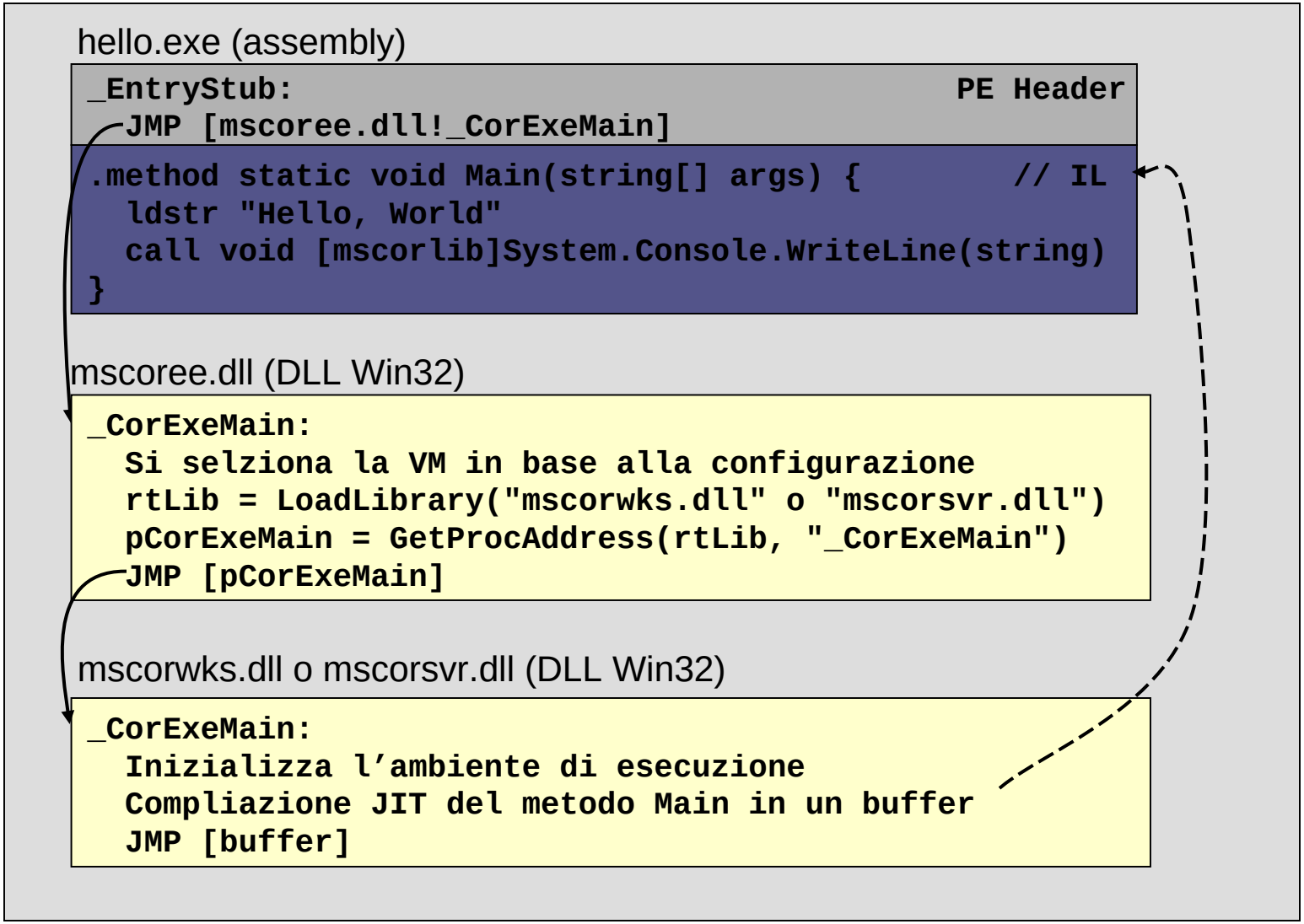
Hello.cs

```
using System;

class Hello {
  public static void Main() {
    Console.WriteLine(“Hello World!");
  }
}
```

```
C:\>csc Hello.cs
C:\>hello
Hello World!
```


Struttura di esecuzione

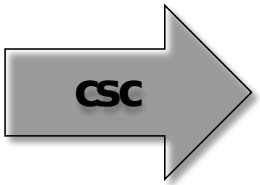


Modello di esecuzione

- Il linguaggio intermedio (IL) viene generato appoggiandosi ad un motore di esecuzione basato su una macchina con stack "infinito"
  - Può essere interpretato simulando tale astrazione o essere compilato *just-in-time* in codice eseguibile di uno specifico processore
- Lo stack contiene lo spazio di valutazione di tutte le espressioni temporanee
  - Le istruzioni IL tipicamente presuppongono che sullo stack sia presenti i propri parametri e depositano qui i loro risultati

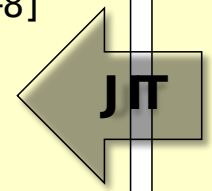
Modello di esecuzione

```
static int Add(int a, int b)
{
  return a + b;
}
```




```
.method static int32 Add(
   int32 a, int32 b)
{
  ldarg.0
  ldarg.1
  add
  ret
}
```

```
7ff8`1c8100d0  mov  dword ptr [rsp+10h],edx
7ff8`1c8100d4  mov  dword ptr [rsp+8],ecx
7ff8`1c8100d8  mov  ecx,dword ptr [rsp+10h]
7ff8`1c8100dc  mov  eax,dword ptr [rsp+8]
7ff8`1c8100e0  add  eax,ecx
7ff8`1c8100e2  jmp  00007ff8`1c8100e4
7ff8`1c8100e4  nop
7ff8`1c8100e5  ret
```




```
.method static int32 Add(
   int32 a, int32 b)
{
  ldarg.0
  ldarg.1
  add
  ret
}
```


Modello di esecuzione

- La compilazione just in time può essere sostituita dal processo NGEN
  - Native image GENeration
  - Compilazione Ahead Of Time, tipicamente eseguita in fase di installazione del codice su una data piattaforma
- I singoli moduli eseguibili (Assembly) vengono trasformati in immagini native per la piattaforma corrente
  - Memorizzate nelle cartelle
    %windir%\assembly\
    NativeImages_v4.0.30319_32

NGEN

```
static void Main() {
  Console.WriteLine(Add(1, 2));
}

static int Add(int a, int b) {
  return a + b;
}
```

```
0:000> !U 00007FFD85E70090
Normal JIT generated code
Arith.Main()
Begin 00007ffd85e70090, size 14
00007ffd`85e70090   sub     rsp,28h
00007ffd`85e70094   mov     ecx,3
00007ffd`85e70099   call    mscorlib_ni+0xd24780 (00007ffd`e4b44780)
   (System.Console.WriteLine(Int32), mdToken: 000000000600099d)
00007ffd`85e7009e   nop
00007ffd`85e7009f   add     rsp,28h
00007ffd`85e700a3   ret
```

1 +2 =3

Confronto

	Win32/COM	.NET
Accesso alla piattaforma	API, oggetti COM	Librerie di classi
Formato del codice	X86	IL
Supporto ai tipi di dati	ad-hoc	Ad oggetti
Gestione della memoria	ad-hoc, conteggio dei rif.	Controllata da un GC
Configurazione	Registry	File XML
Gestione delle versioni	ad-hoc	integrata
Origine del codice	Disco	Disco, rete
Unità di distribuzione	DLL, EXE	Assembly
Unità di esecuzione	Processo	AppDomain
Sicurezza	Basata sui ruoli	Basata sui ruoli e sull'evidenza
Accesso alla rete	Socket, DCOM	Socket, remoting, servizi web, WCF
Gestione degli errori	HRESULT, eccezioni, GetLastError()	Modello di eccezioni unificato

Sintassi del linguaggio: commenti XML

```
class Element
{
   /// <summary>
   ///    Returns the attribute with the given name and
   ///    namespace</summary>
   /// <param name="name">
   ///    The name of the attribute</param>
   /// <param name="ns">
   ///    The namespace of the attribute, or null if
   ///    the attribute has no namespace</param>
   /// <return>
   ///    The attribute value, or null if the attribute
   ///    does not exist</return>
   /// <seealso cref="GetAttr(string)"/>
   ///
   public string GetAttr(string name, string ns) {
      ...
   }
}
```

Sintassi del linguaggio: istruzioni ed espressioni

- Sostanzialmente simili a C/C++/Java
  - L'istruzione **switch (…)** non ha il comportamento *fall-through*
  - Non sono ammessi salti all'interno di blocchi
  - Introdotta l'istruzione **foreach (…)** per iterare su array e classi che implementano l'interfaccia System.IEnumerator<T>
  - È possibile controllare la generazione di *overflow* nelle espressioni mediante i costrutti **checked/unchecked**
  - Uso della parola chiave **var** per dedurre il tipo di una variabile dal valore che le viene assegnato

Sintassi del linguaggio: l'istruzione foreach

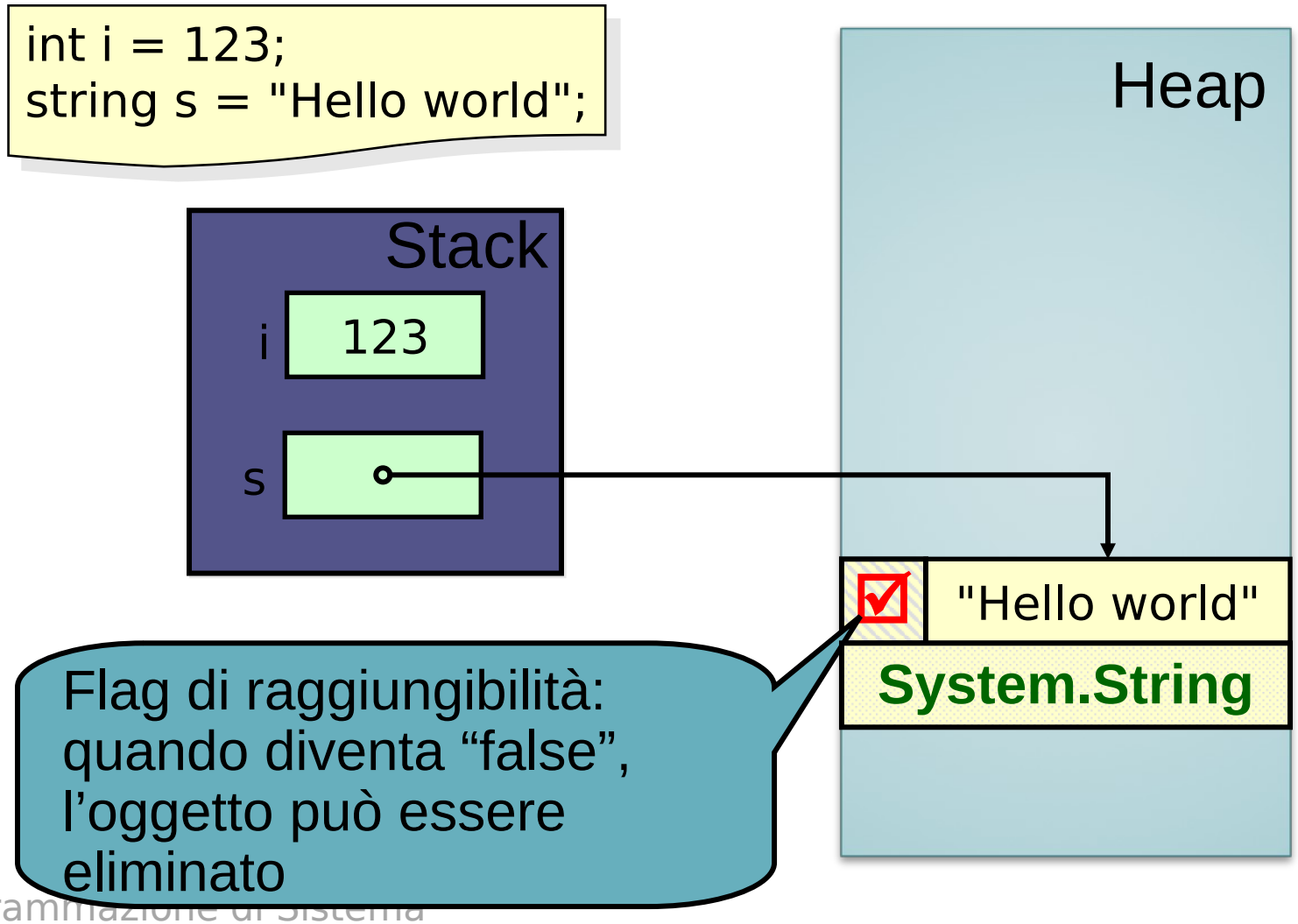
```
public static void Main(string[] args) {
   foreach (string s in args) Console.WriteLine(s);
}
```

```
IList customers =...;
foreach (Customer c in customers.OrderBy("name")) {
   if (c.Orders.Count != 0) {
      ...
   }
}
```

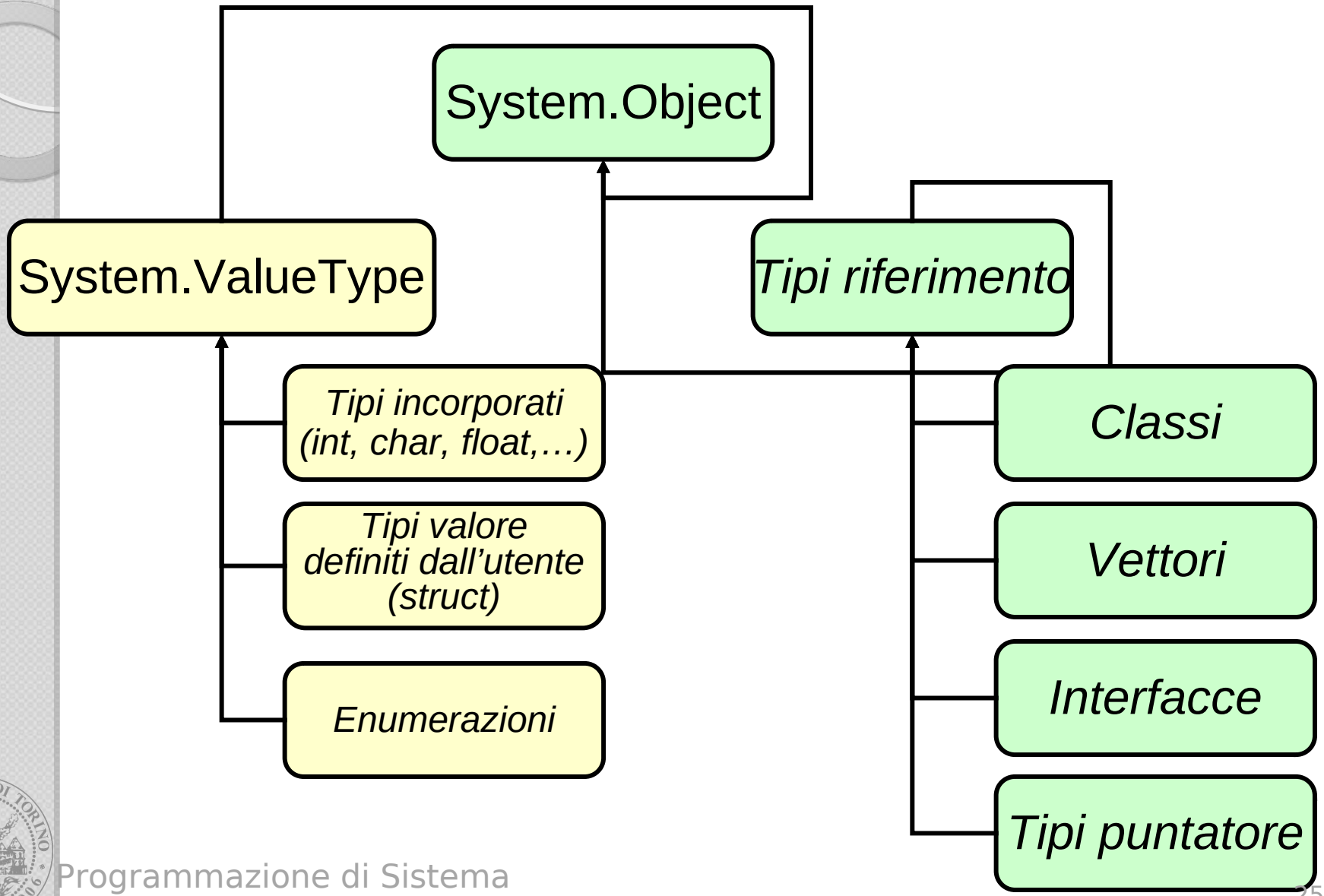

Tipi di dato

- Tutti i dati utilizzati in C# hanno un tipo
  - Il tipo definisce al gamma di valori consentita e l'insieme di operazioni lecite su un determinato dato
  - Tutti i tipi sono organizzati in una gerarchia di ereditarietà, la cui radice è System.Object
- Tipi valore:
  - Contengono direttamente il dato
  - Non possono valere "null"
  - Quando vengono copiati, si effettua una copia del valore
- Tipi riferimento:
  - Contengono un puntatore al valore
  - Il valore si trova sull'*heap gestito*
  - Possono valere "null"
  - Quando vengono copiati, si effettua una copia del puntatore

Tipi valore/riferimento



Organizzazione dei tipi

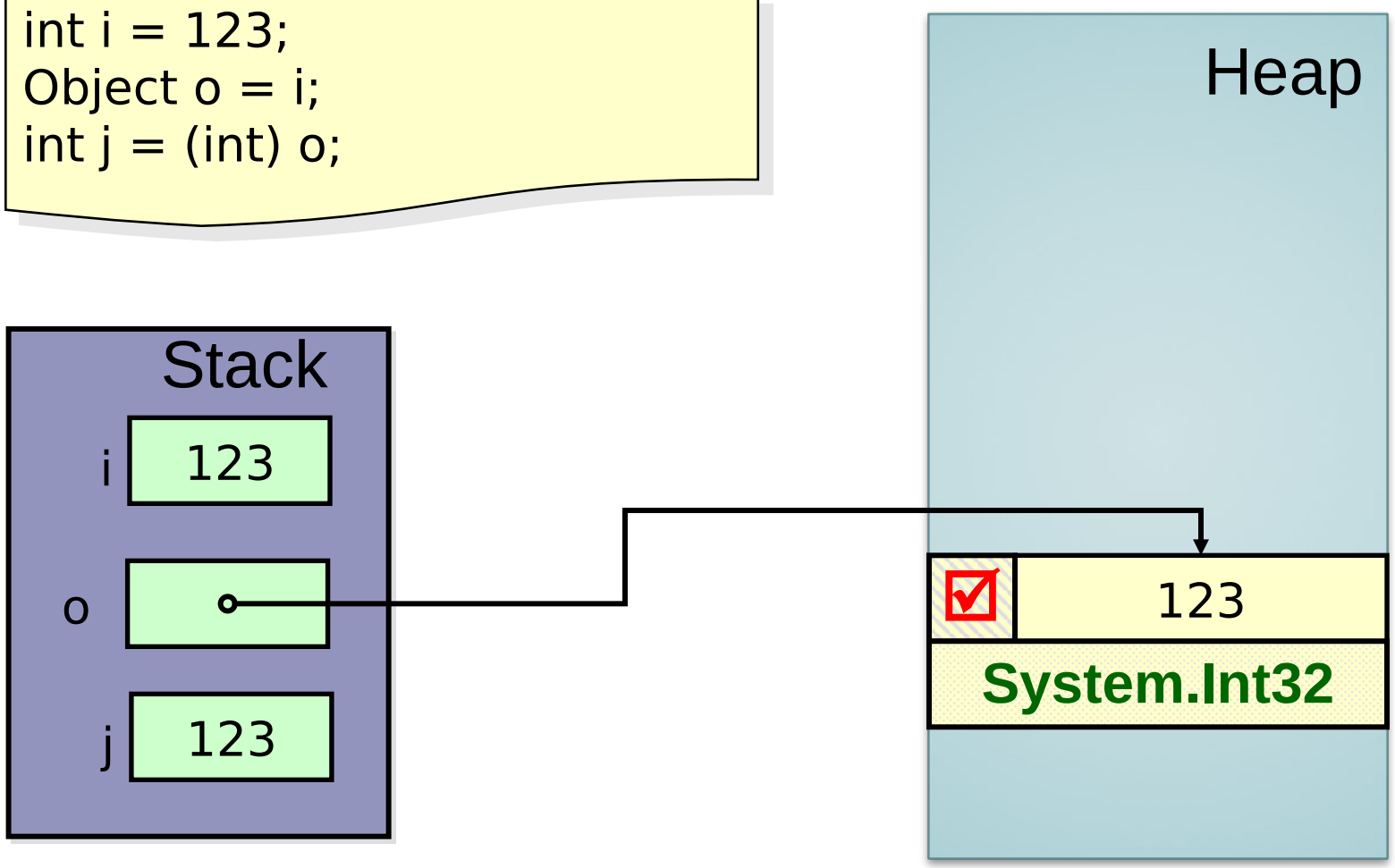


Boxing/unboxing (1)

- Anche i tipi valore hanno un corrispondente tipo riferimento
  - Il compilatore automaticamente converte tra i due formati quando necessario
  - Si facilita l'utilizzo di classi generiche (array, liste, ...) con tipi elementari
- Boxing
  - Trasformazione da valore a riferimento
  - Allocazione automatica dello spazio
- Unboxing
  - Trasformazione da riferimento a valore
  - Viene eseguito un controllo sul tipo: l'ambiente di esecuzione genera un'eccezione in caso di incompatibilità

Boxing/unboxing (2)

```
int i = 123;
Object o = i;
int j = (int) o;
```



Tipi predefiniti

- Numerici interi
  - Con segno: **sbyte**, **short**, **int**, **long**
  - Senza segno: **byte**, **ushort**, **uint**, **ulong**
- Numerici reali
  - **float**, **double**, **decimal**
- Non numerici
  - **char** (formato Unicode), **bool**
- Riferimento
  - **object** base di tutti i tipi (System.Object)
  - **string** sequenza immutabile di caratteri Unicode (System.String)

Strutture

- Dati aggregati simili alle classi
  - Possono avere campi, metodi, costruttori
  - Non supportano l'ereditarietà, solo l'implementazione di interfacce
- Informazioni di tipo valore
  - Allocate sullo stack, e non sull'heap
  - La chiamata al costruttore comporta la sola inizializzazione dei campi, non l'acquisizione di memoria dallo heap gestito
  - La copia comporta la replica dei dati
  - Permettono una gestione più efficiente della memoria

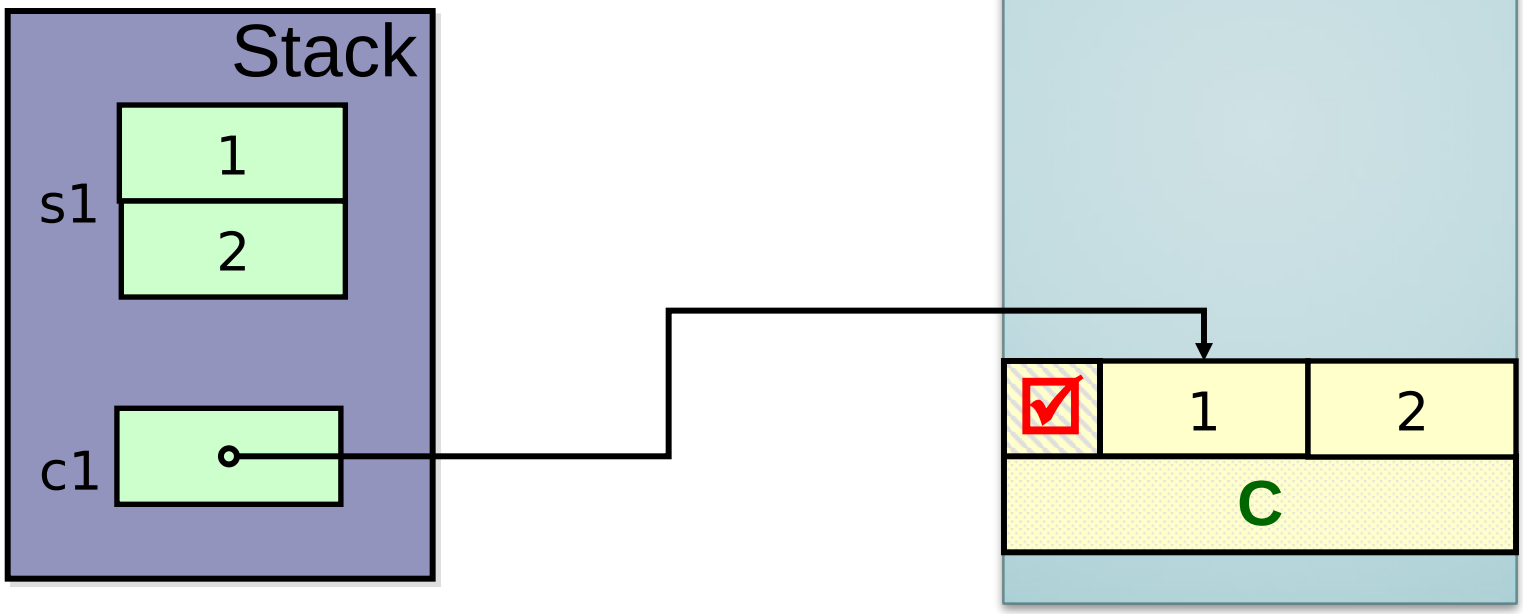
Classi

- Organizzate in una gerarchia di ereditarietà semplice
  - Ogni classe può però implementare molte interfacce
- Identificazione
  - Spazio dei nomi: identifica il “cognome” della classe, analogo al concetto di “package” in java
  - Nome: identifica la classe all’interno del proprio spazio
- Quattro modificatori di visibilità:
  - **public, protected, private, internal**
- Elementi delle classi
  - Costanti, campi, metodi, proprietà, indicizzatori, eventi, operatori, costruttori, distruttori
  - Relativi alle singole istanze o all’intera classe (**static**)

Classi e strutture

```
struct S {int a,b;...}
class C {int a,b;...}

S s1=new S(1,2);
C c1=new C(1,2);
```



Campi e costanti

- Campi
  - Variabili legate alle singole istanze
  - public class Color { int r,g,b; }
- Costanti
  - Campi preceduti dalla parola chiave **const**
  - Non possono essere modificati dopo l'inizializzazione
  - public class Math {
    const static double pi=3.1415928;
    }

Metodi

- Funzioni che fanno riferimento all'istanza della classe
  - Possono avere parametri (anche in numero variabile) ed un tipo di ritorno
  - public class Color {
    int r,g,b;
    void reset( ) {
    r=0; g=0; b=0;
    }
    }

Interfacce

- Astrazioni del comportamento di un oggetto
  - Possono definire metodi, proprietà, indicizzatori ed eventi
- Ogni classe può implementare molte interfacce
  - Occorre fornire il codice corrispondente a tutte le funzioni dichiarate dalle singole interfacce
- ```
  interface IDataBound {
    void Bind(IDataBinder binder);
  }

  class EditBox: Control, IDataBound {
    void IDataBound.Bind(IDataBinder binder) {...}
  }
  ```



  ```

Enumerazioni

- Tipi di dato con dominio dichiarato in modo esplicito
  - Ogni valore viene implementato da un numero
  - Sono supportate operazioni tra valori dello stesso tipo (+, -, ++, --, &, |, ^, ~)
  - È possibile indicare la quantità di memoria che occorre allocare
  - A differenza di altri linguaggi, i valori non sono convertibili automaticamente in interi
-
```
enum Color: byte {
   Red     = 1,
   Green   = 2,
   Blue    = 4,
   Black   = 0,
   White   = Red | Green | Blue,
}
```


Proprietà

- Campi virtuali di un oggetto per i quali si esplicitano le operazioni di assegnazione e lettura
  - Permettono di utilizzare una sintassi naturale, preservando il controllo sul codice generato
  - Normalmente si appoggiano su una variabile privata dello stesso tipo (che può essere creata automaticamente dal compilatore)

```
public class Button: Control {
   private string _caption;
   public string Caption {
      get {return _caption;}
      set {
         _caption = value;
         Refresh();
      }
   }
}
```

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

```
public class Test {
  public string Caption {
     get; set;
  }
}
```

Indicizzatori

- Array virtuali associati ad un oggetto, per i quali si esplicitano le operazioni di accesso in lettura e scrittura alle singole celle
  - Gli indici possono essere non numerici
  - Sono possibili versioni *overloaded* dello stesso indicizzatore
  - Sono possibili indici multidimensionali

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

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

Callback e delegati

- L'esecuzione di un algoritmo può richiedere che un metodo chiamato richiami un metodo del chiamante
  - Perché questo avvenga in modo parametrico, occorre che il chiamato conosca sia l'identità del chiamante che il metodo da invocare (con i relativi parametri)
- L'implementazione del pattern "callback" richiede una certa quantità di codice
  - In C/C++ questo si gestisce tramite liste di puntatori a funzioni ed ai relativi parametri
  - In Java si utilizzano interfacce "listener", oggetti di tipo lista, metodi per registrare e cancellare gli ascoltatori e codice per iterare le notifiche

Delegati

- Tipi di dato che incapsulano il pattern di chiamata a *callback*
  - Modellano liste di coppie (oggetto, metodo) da invocare a richiesta
- Si utilizzano per definire variabili locali, campi e/o parametri
  - Tali variabili contengono liste di istanze del delegato create mediante l'operatore *new*
  - Gli operatori =, += e -= permettono di manipolare il contenuto associato a tali variabili
- Tali variabili possono essere invocate come metodi
  - Per ogni oggetto presente nella lista, invocano il metodo relativo, passando i parametri ricevuti
  - In caso di eccezione, il procedimento si arresta

Uso di delegati

1. Si dichiara il tipo delegato
   - delegate void **Handler**(string msg);
2. Si dichiara una variabile (campo o parametro) avente il tipo del delegato
   - Handler myHandler;
3. Si assegna a tale variabile uno o più valori
   - myHandler = new **Handler**(myObj.someMethod);
   - myHandler += new **Handler**(anotherObj.doSomething);
   - myHandler += evenAnotherObject.methodName;
4. Si invoca il delegato
   - myHandler("Message description");

Eventi (1)

- I delegati implementano gli elementi base necessari al pattern *callback*
  - Un'assegnazione impropria su una variabile di tipo *delegate* potrebbe cancellare possibili destinatari
  - La parola chiave *event* applicata ad una variabile *delegate* ne restringe l'utilizzo ai soli operatori += e -= (*add* e *remove*)
  - Solo il possessore dell'evento può invocare il delegato relativo (*raise*)
- I delegati associati ad un evento hanno tipicamente due parametri e ritornano void
  - Il primo parametro, di classe *System.Object*, rappresenta il mittente dell'evento
  - Il secondo parametro, di classe *System.EventArgs*, rappresenta gli eventuali dettagli associati all'evento

Eventi (2)

```
public delegate void Handler(object sender, EventArgs e);
public class Button
{
  public event Handler Click;
  protected void OnClicked(...) {
    var clicked = Click; //in caso di accesso multithread
    if (clicked!=null)
      clicked(this,new MouseEventArgs(...));//Solleva l'evento
  }
}

public class Test
{
  public static void MyHandler(object sender, EventArgs e) {
    // React to event...
  }
  public static void Main() {
    Button b=new Button();
    b.Click += new Handler(MyHandler);
    ...
  }
}
```

Eventi (3)

```
class Button {
  private Handler _clicked;
  public event Handler Clicked {
    add {
      Action old, @new;
      do {
        old = _clicked;
        @new = old + value;
           // chiama
Delegate.Combine(old,value)
      } while (Interlocked.CompareExchange(
                  ref _clicked, @new, old) !=
old);
    }
    remove { ... } }
}
```

Funzioni lambda (1)

- Ad un'istanza di delegato (o di evento) è possibile assegnare anche un'espressione od una funzione lambda
- Si usa rispettivamente la sintassi
  - **(** <paramertri> **) =>** <espressione>
  - **(** <paramertri> **) => {** <istruzioni> **; }**
- L'uso delle parentesi per delimitare i parametri è facoltativo se c'è un solo parametro
- Se nel corpo della funzione/espressione sono presenti variabili definite al suo esterno, queste vengono catturate per riferimento
  - Il ciclo di vita di tali variabili, viene automaticamente prolungato fino a che ne esiste un riferimento valido

Funzioni lambda (2)

```
delegate bool D1();
delegate bool D2(int i);
class Test {
  D1 del1;
  D2 del2;

  public void method(int input) {
    int j=0; // j è una variabile locale inizializzata
    del1 = () => { j=10; return j>input; }
    del2 = (x) => { return x==j; }
    bool result = del1(); // true , j diventa 10
  }

  public static void Main() {
    Test test=new Test();
    test.method(5);
    bool result = test.del2(10); // true, j vale 10
  }
}
```


Attributi (1)

- Annotazioni del codice sorgente accessibili durante l'esecuzione mediante *reflection*
  - Permettono di associare informazioni alle classi o ai loro elementi (metadati)
  - Introducono meccanismi flessibili per la gestione del codice senza richiedere soluzioni esterne (come file IDL)
  - Vengono memorizzati in istanze della classe **System.Attribute** o in classi da essa derivate
  - Il loro tipo viene controllato in fase di compilazione
- Usi tipici
  - URL della documentazione di una classe
  - Modello di esecuzione concorrente
  - Modellazione della serializzazione in XML
  - Servizi Web
  - …

Attributi (2)

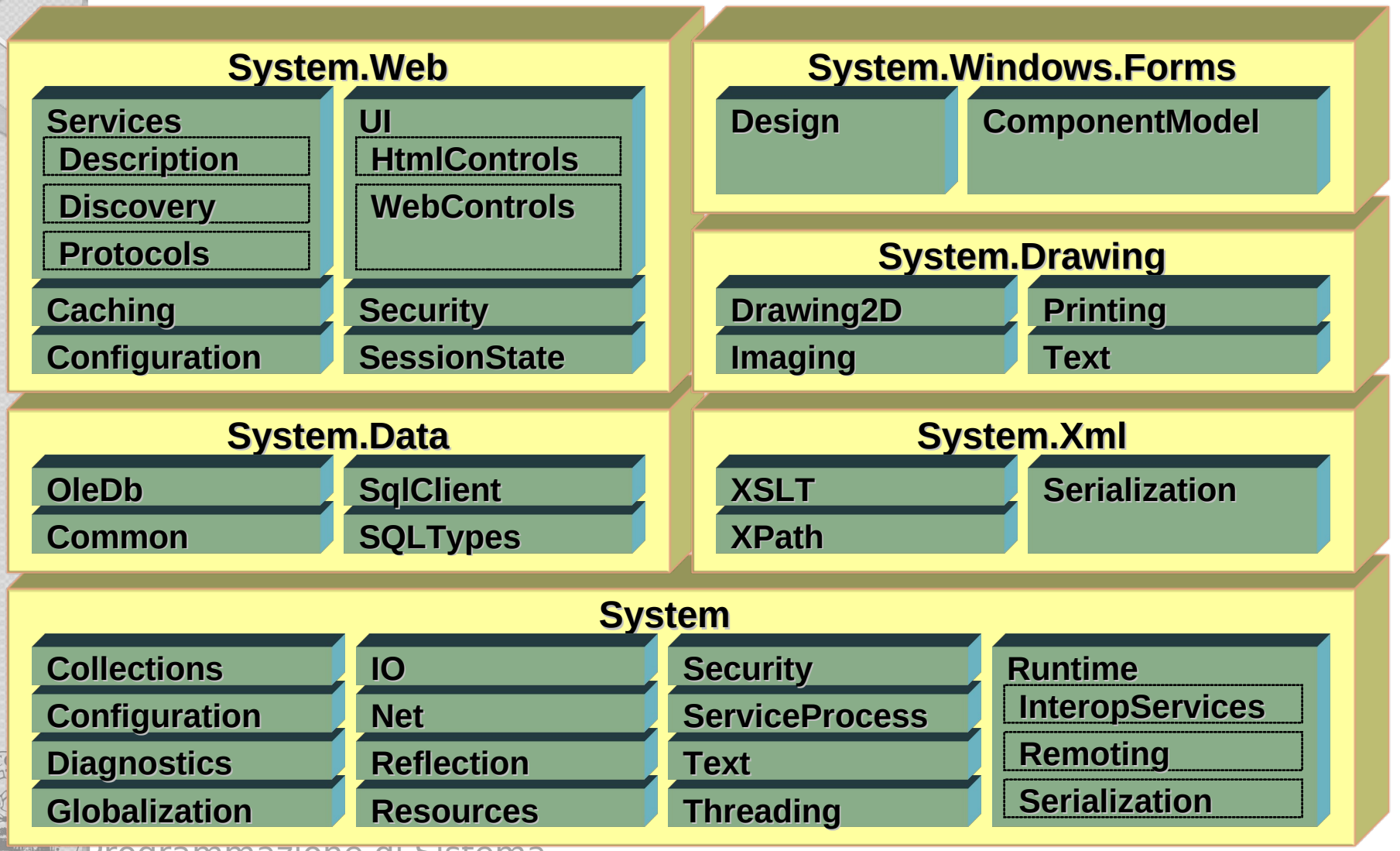
```
public class OrderProcessor
{
  [WebMethod]
  public void SubmitOrder(PurchaseOrder order) {...}
}

[XmlRoot("Order", Namespace="urn:acme.b2b-schema.v1")]
public class PurchaseOrder
{
  [XmlElement("shipTo")]  public Address ShipTo;
  [XmlElement("billTo")]  public Address BillTo;
  [XmlElement("comment")] public string Comment;
  [XmlElement("items")]   public Item[] Items;
  [XmlAttribute("date")]  public DateTime OrderDate;
}
public class Address {...}
public class Item {...}
```

La libreria del framework

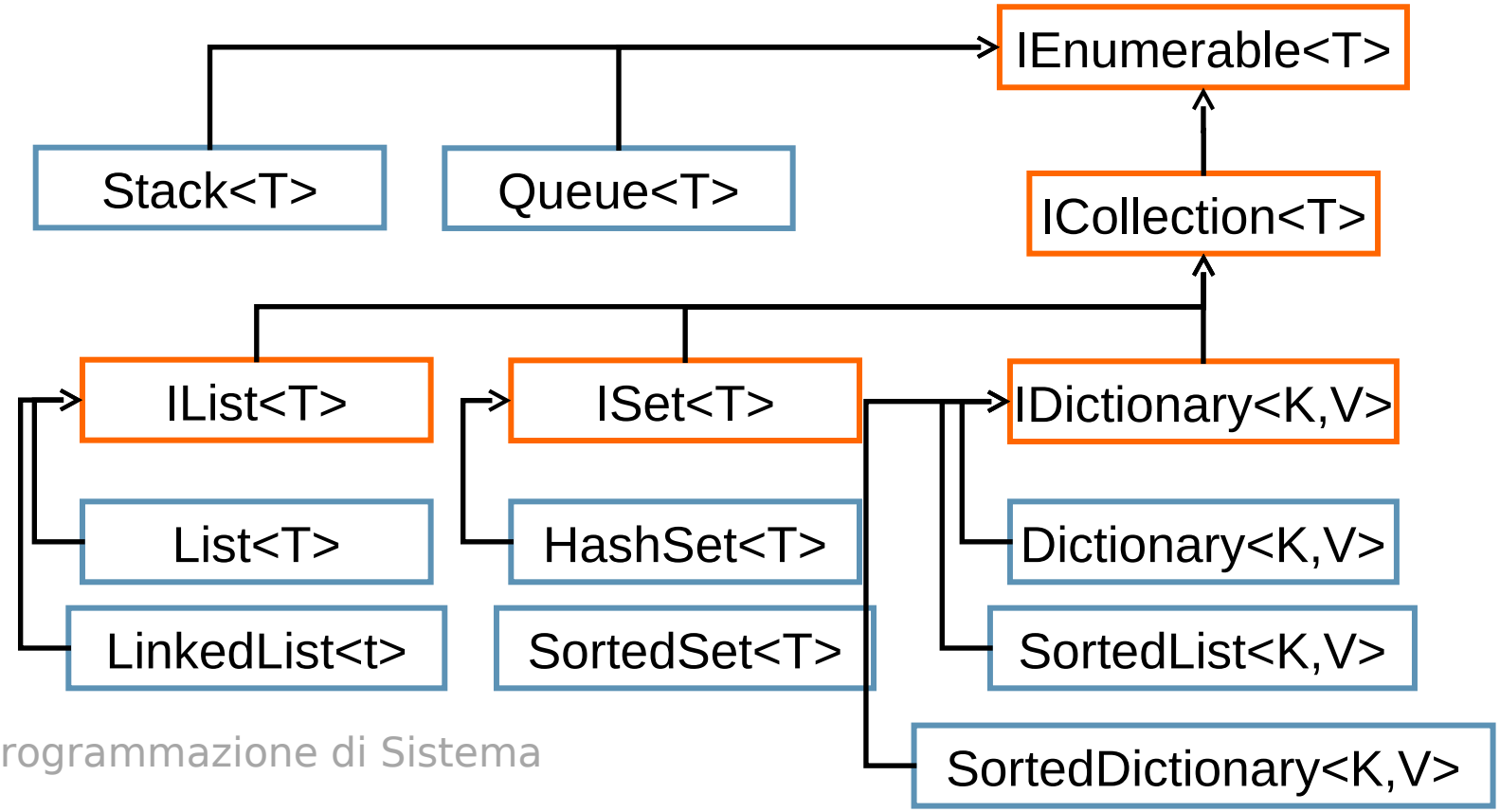
- API complessa ed articolata
  - Composta da più di 7000 tra classi, strutture, interfacce, enumerazioni e delegati
  - Circa 100 *namespace* organizzati gerarchicamente
  - La radice di tutte le classi è System.Object
- Molte le aree funzionali
  - Gestione di collezioni di oggetti
  - I/O legato a file e flussi
  - Manipolazione di espressioni regolari
  - Interazioni con la rete
  - Accesso ai dati
  - Riflessione
  - Interfacce grafiche
  - …

.NET Class Library



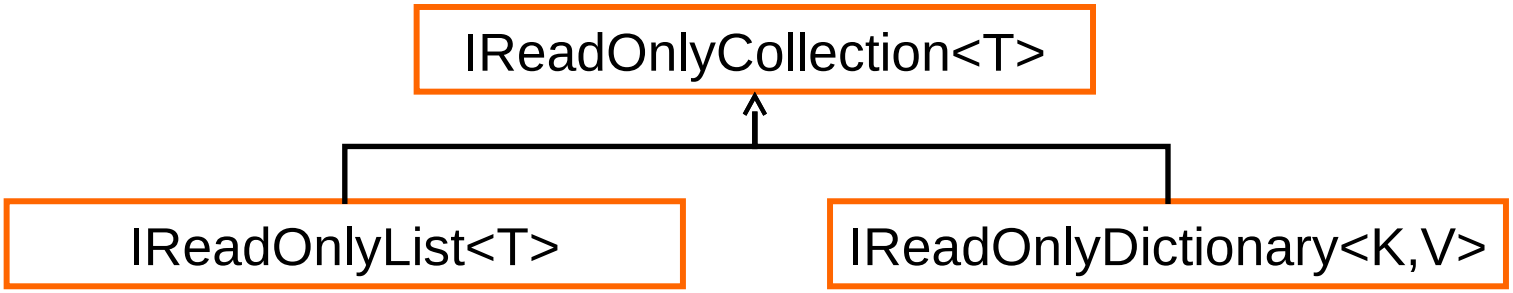
Collezioni generiche

- Permettono l'uso di dati fortemente tipati attraverso il meccanismo di *type-erasure*
  - Definite nello spazio dei nomi System.Collections.Generics



Collezioni in sola lettura

- Insieme di interfacce che escludono la possibilità di modificare i dati contenuti in una collezione
  - Implementate attraverso l'uso di un wrapper
  - In caso di cambiamento della collezione originale, i dati della collezione in sola lettura cambiano



Collezioni osservabili

- Offrono eventi per annunciare l'inserimento, la cancellazione, lo spostamento e la sostituzione di elementi al loro interno
  - Definite nello spazio dei nomi System.Collections.ObjectModel

ObservalbleCollection<T>

Collezioni concorrenti

- Offrono meccanismi per gestire, in modo thread-safe, collezioni di dati
  - Definite nello spazio dei nomi System.Collections.Concurrent

BlockingCollection<T>

ConcurrentBag<T>

ConcurrentDictionary<K,V>

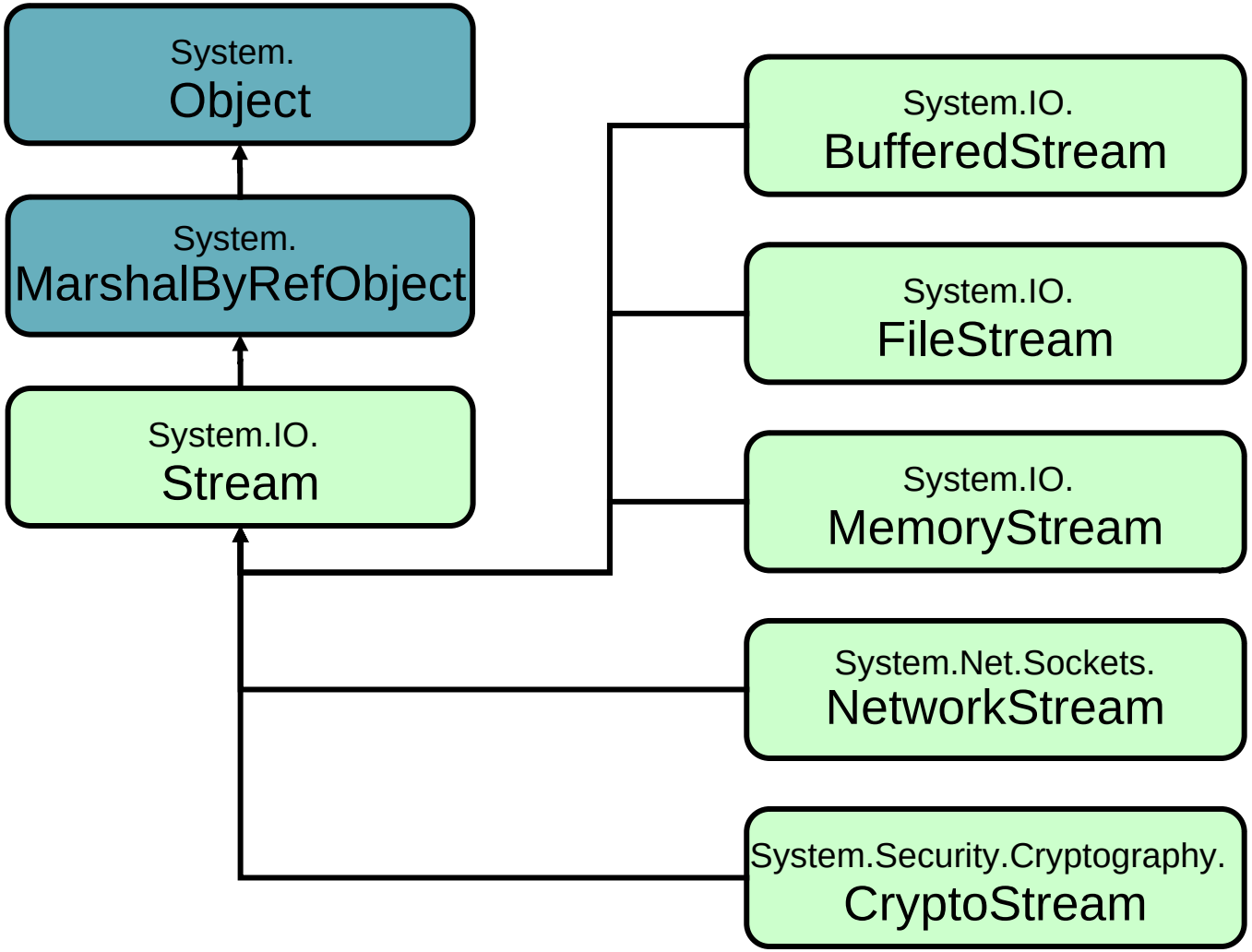
ConcurrentQueue<T>

ConcurrentStack<T>

Input/Output

- .NET offre un ricco insieme di classi per eseguire operazioni I/O
  - Principalmente contenuto nel *namespace* System.IO
  - Estremamente raffinato e complesso se paragonato a *<Stdio.h>*
- La classe *Stream* costituisce la principale astrazione
  - Rappresenta una sequenza di byte
  - Offre i meccanismi di base per la lettura, la scrittura, il posizionamento all'interno di tale sequenza
  - Classe astratta: non può essere istanziata

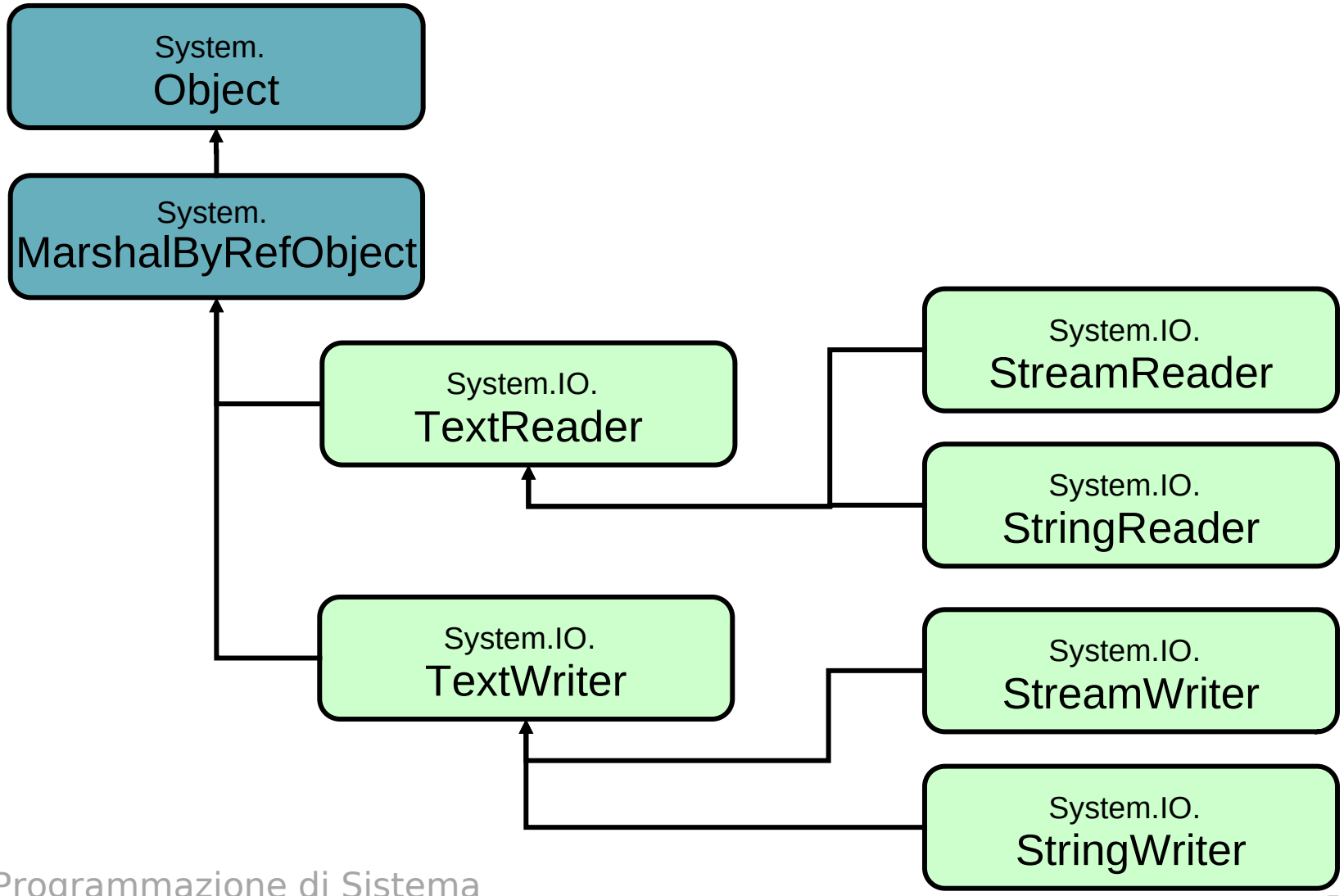
Tipologie di flussi



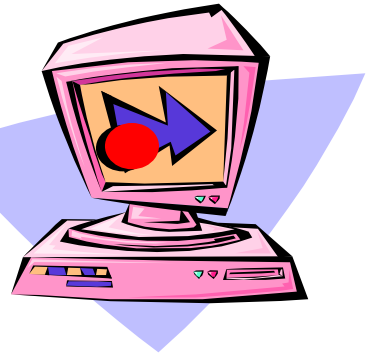
Lettura e scrittura

- Le operazioni supportate dalla classe Stream riguardano la lettura e la scrittura di byte
- Se occorre leggere/scrivere caratteri occorre (de)codificarli
  - Possibili molti schemi: UTF-7, UTF-8, UTF-16,...
- Se occorre leggere/scrivere strutture binarie più articolate occorre (de)serializzarle
  - Molte più alternative...

Operare con il testo

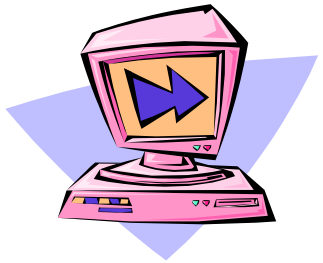


Interfacce grafiche (I)



- A differenza di quanto accade nei programmi con interfaccia a caratteri, non c'è un unico flusso di comunicazione tra utente e programma
- Esiste uno spazio bidimensionale in cui sono **disegnati** gli "**strumenti**" necessari all'interazione
  - La comunicazione da e verso l'applicazione avviene direttamente utilizzando gli oggetti presenti sullo schermo
  - L'utente utilizza la tastiera e un dispositivo di puntamento (mouse) con cui indica e comanda l'applicazione

Interfacce grafiche (II)



- Ciascun oggetto di ingresso (bottoni, liste, caselle di testo, …) offre un numero limitato di alternative
  - Non è necessario introdurre analisi lessicale e sintattica del testo in ingresso
  - Non è noto però a priori in quale ordine avvenga l'interazione dell'utente con i diversi canali
- La struttura dei programmi muta radicalmente
  - Invece di richiedere l'esecuzione di una sequenza di operazioni, un programma deve adattarsi a **reagire** ad eventi esterni, cooperando con il sistema operativo
- Oltre ad implementare la propria logica interna, un'applicazione deve
  - Scegliere e comporre gli strumenti che formano le videate
  - Rispondere alle richieste dell'utente aggiornando coerentemente quanto visualizzato

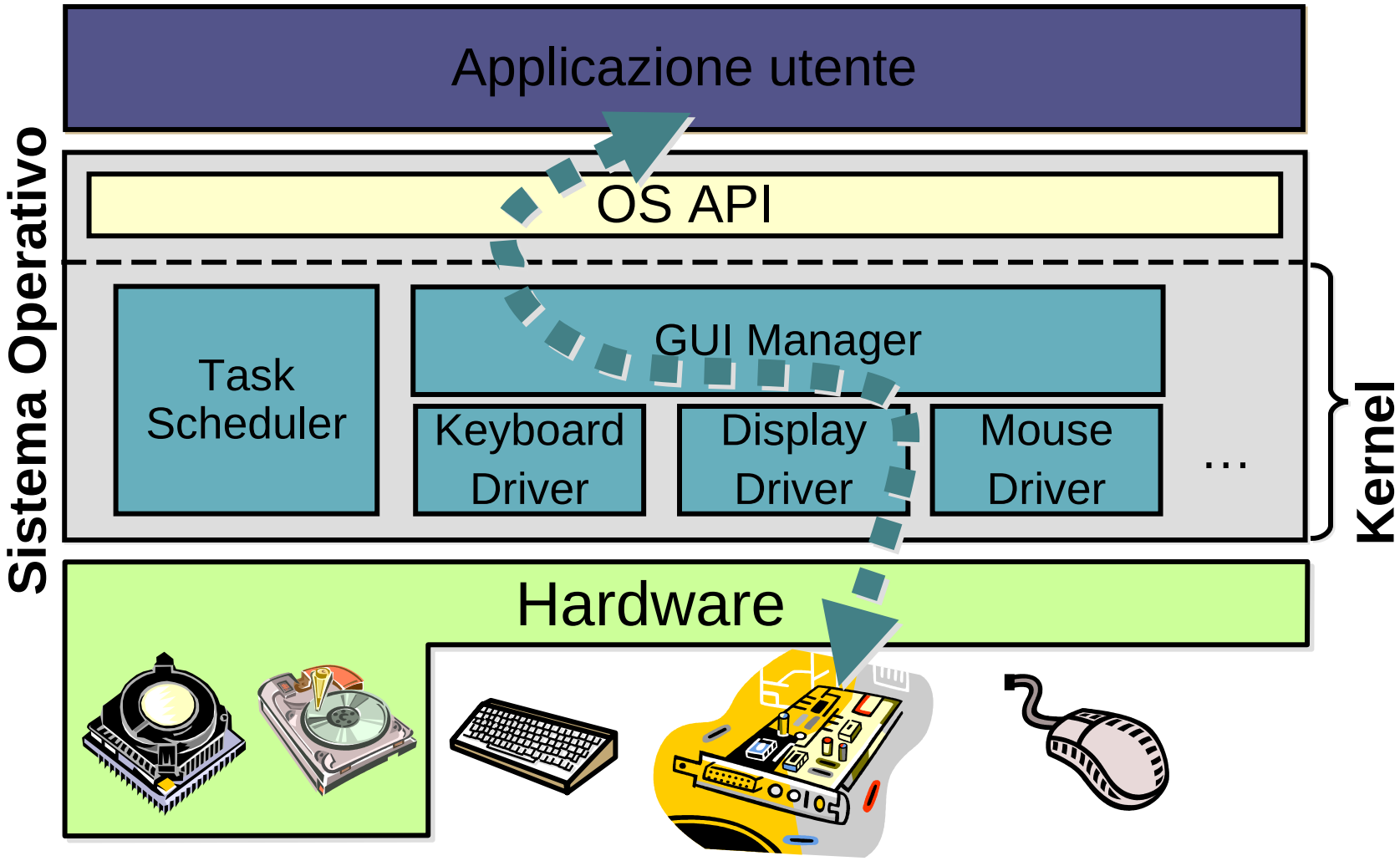
Interfacce grafiche (III)

- Mouse, tastiera e schermo sono risorse condivise da molte applicazioni contemporaneamente
  - Nessuna interagisce direttamente con essi
  - Il sistema operativo ne scherma l'accesso offrendo opportuni meccanismi per consentire all'applicazione di interagire con tali periferiche
- Per interagire con lo schermo, un'applicazione crea una o più finestre
  - Porzioni logiche di schermo in cui il programma può disegnare
  - Ogni finestra ha un identificativo opaco univoco (handle) mantenuto dal sistema operativo
- Per interagire con mouse, tastiera, altre applicazioni, il sistema operativo mette a disposizione una coda di messaggi
  - Da essa l'applicazione attinge informazioni circa gli eventi che la riguardano e reagisce corrispondentemente

GUI Manager (1)

- Componente software incaricato di gestire direttamente le periferiche e interagire con le applicazioni
  - Identifica “eventi significativi” e li notifica alle applicazioni sotto forma di opportune strutture dati (messaggi)
  - Gestisce lo smistamento dei messaggi verso le diverse applicazioni, identifica e risolve i conflitti
  - Permette l’accesso alle risorse grafiche

GUI Manager (2)



Programmazione reattiva

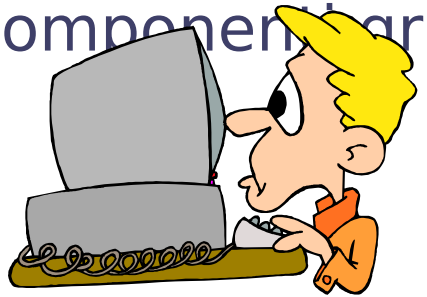
- Nei programmi di tipo GUI, non è prevedibile quale azione compia l'utente in quale momento
- È necessario predisporre un insieme di azioni da compiere quando si verifica un certo evento
  - Il programma "reagisce" agli eventi esterni invocando una **breve** procedura che
    - Aggiorna lo stato interno del programma
    - Richiede al GUI Manager l'aggiornamento della rappresentazione grafica

Struttura delle applicazioni

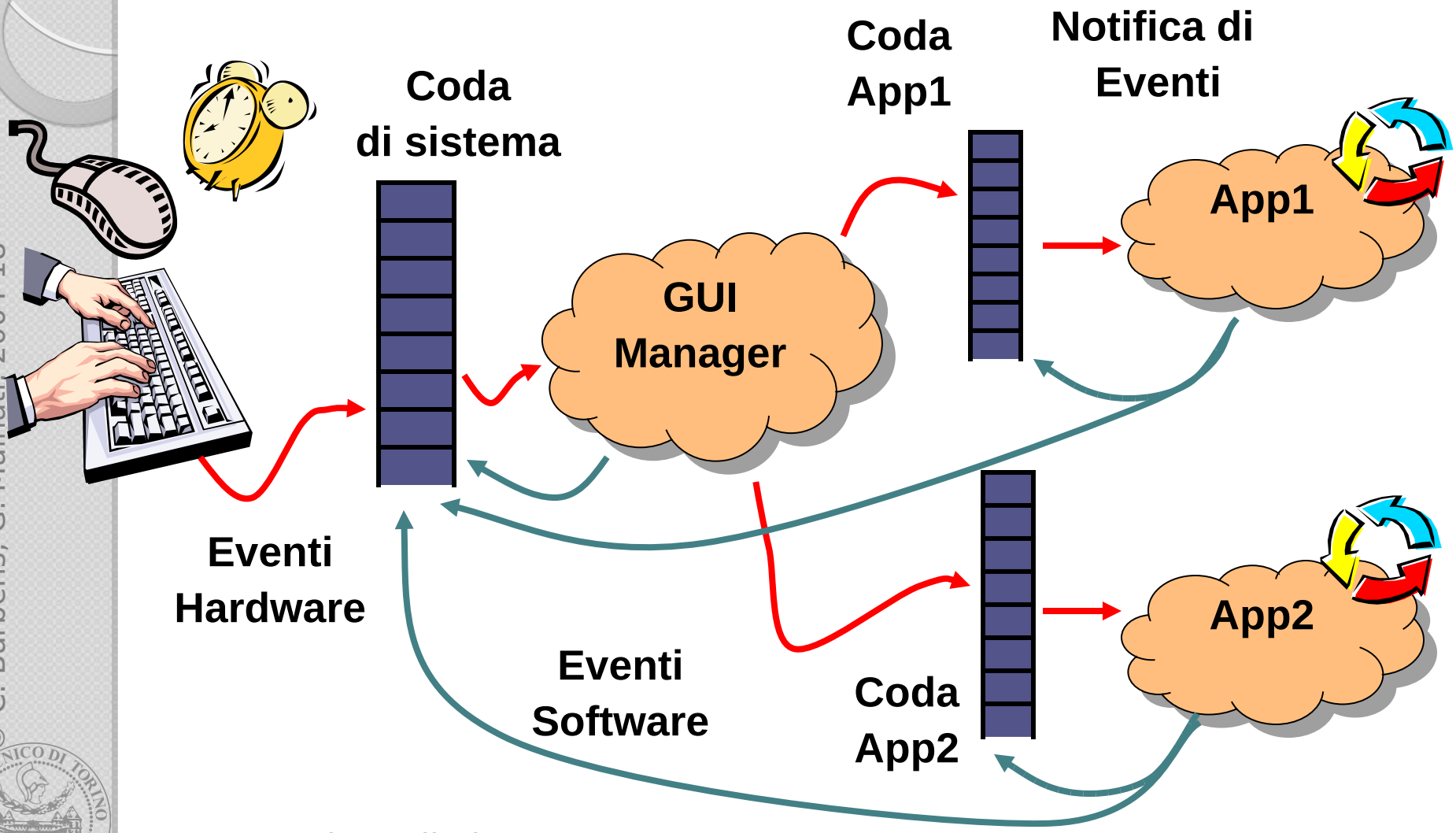
- Per interagire con il GUI Manager un'applicazione deve
  - Iniziare una sessione di lavoro, creando la propria coda di messaggi
  - Richiedere la creazione di risorse grafiche (finestre, bottoni, campi di testo, …)
  - Predisporre, per ogni widget ed evento atteso, un'opportuna routine di callback
  - Iterare sulla coda dei messaggi, inoltrando le richieste ricevute alla relativa callback
- Le modalità con cui queste operazioni vengono effettuate, dipendono dal sistema operativo e/o dagli eventuali strati software intermedi adottati

Struttura di un programma

- Inizializzazione dell'interfaccia
  - Creazione della finestra principale e delle sue sottofinestre, indicando il comportamento che esse dovranno avere
- Attesa e reazione agli eventi
  - Si attende ciclicamente che il GUI Manager depositi un messaggio nella coda che è stata creata per l'applicazione e si esegue l'azione corrispondente
  - La libreria del framework traduce tali messaggi in invocazioni degli eventi corrispondenti nei diversi componenti grafici



Flusso dei messaggi



Windows form (1)

- Modello di programmazione integrato per lo sviluppo di applicazioni standard Win32
  - .NET mette a disposizione un insieme ricco e unificato di funzioni grafiche e di controllo per tutti i linguaggi
- Estendono la classe System.Windows.Forms.Form

Applicazioni grafiche

- Modello di esecuzione reattiva a thread singolo
  - Per garantire la corretta consegna degli eventi, occorre invocare il metodo statico **Application.Run(…)** che incapsula il loop dei messaggi
- Molteplici opzioni per realizzare l'interfaccia grafica
  - Primitive GDI+ per l'accesso al video
  - Gestione dell'input tramite eventi legati a menu, mouse e tastiera
- Ampia rassegna di controlli predefiniti con cui personalizzare l'interfaccia
- Supporto avanzato per la realizzazione di finestre di dialogo

Controlli grafici

- Tutti i componenti grafici derivano dalla classe System.Windows.Forms.Control
  - Essa definisce le proprietà, i metodi e gli eventi comuni ai componenti visuali: dimensioni, visibilità, posizione, organizzazione gerarchica, colore, font, …
  - Gli eventi legati alla classe scandiscono il ciclo di vita di un componente grafico, informando i potenziali ascoltatori di ogni interazione e cambiamento di stato legato all'interazione del componente con l'ambiente di presentazione e/o l'utente finale

Le principali proprietà della classe Control

- Posizione e dimensioni (in pixel)
  - Left, Right, Top, Bottom, Width, Height, Size, MinimumSize, MaximumSize, Padding, Margin,...
- Aspetto
  - BackColor, BackgroundImage, ForeColor, Font, Text, Cursor,...
- Organizzazione logica
  - Parent, Controls
- Interattività
  - Enabled, Visible

Interfacce grafiche

- Un'interfaccia è costituita da un albero di oggetti grafici la cui radice è costituita da un'istanza della classe Form
  - Gli oggetti figli sono accessibili tramite la proprietà Controls (di tipo ControlCollections)
- L'albero viene costruito programmaticamente
  - VisualStudio offre la possibilità di "disegnare" l'interfaccia grafica sintetizzando il codice corrispondente alla rappresentazione visuale costruita dal programmatore
  - Ad ogni componente viene associato un campo della classe che estende Form
  - Il codice viene posto nel metodo "InitializeComponents()" che viene salvato in un file nascosto per evitare che sia manipolato direttamente dal programmatore

Aggiungere l'interattività

- È possibile rendere interattivo l'albero dei componenti registrando ascoltatori sugli eventi corrispondenti
  - Interazione con il mouse (Click, DoubleClick, DragEnter, DragOver, DragLeave, DragDrop, Enter, Leave, …)
  - Interazione con la tastiera (KeyDown, KeyUp, KeyPress, GotFocus, LostFocus, …)
  - Interazione con il sistema di layout e visualizzazione (Paint, Print, Resize, Layout, …)

Interattività e azioni

- Le azioni svolte all'interno dei metodi delegati alla gestione degli eventi devono essere di breve durata per non bloccare l'interfaccia grafica
  - Se l'azione dura a lungo, occorre delegarne l'esecuzione ad un thread secondario (od al thread pool)
- E' importante notare che i metodi (le proprietà, gli indicizzatori, gli eventi,..) di un componente grafico possono essere manipolati solo nel contesto del thread che li ha creati
  - Fa eccezione il metodo Invoke(…): questo può essere chiamato da un thread secondario per richiedere l'esecuzione di un metodo da parte del thread principale

Grafica 2D

- L'evento Paint offre l'accesso ad un contesto grafico GDI+
  - Attraverso la proprietà Graphics dell'oggetto PaintEventArgs che viene notificato al gestore
  - Tramite esso è possibile utilizzare tutte le funzionalità di disegno sullo schermo, tracciamento di scritte e manipolazione di immagini offerte dalla libreria nativa

HelloForm.cs

```
using System;
using System.Windows.Forms;
using System.Drawing;

class HelloForm: Form {
  HelloForm() {
    Text = “HelloForm”;
    Paint+= PaintWindow;
  }
  private void PaintWindow(Object s,PaintEventArgs e) {
    e.Graphics.DrawString(“Hello Windows Form”, Font,
      new SolidBrush(Color.Black), ClientRectangle);
  }
  public static void Main() {
    Application.Run(new HelloForm());
  }
}
```