Introduzione al C#

Raffaele Cappelli
raffaele.cappelli@unibo.it

Contenuti

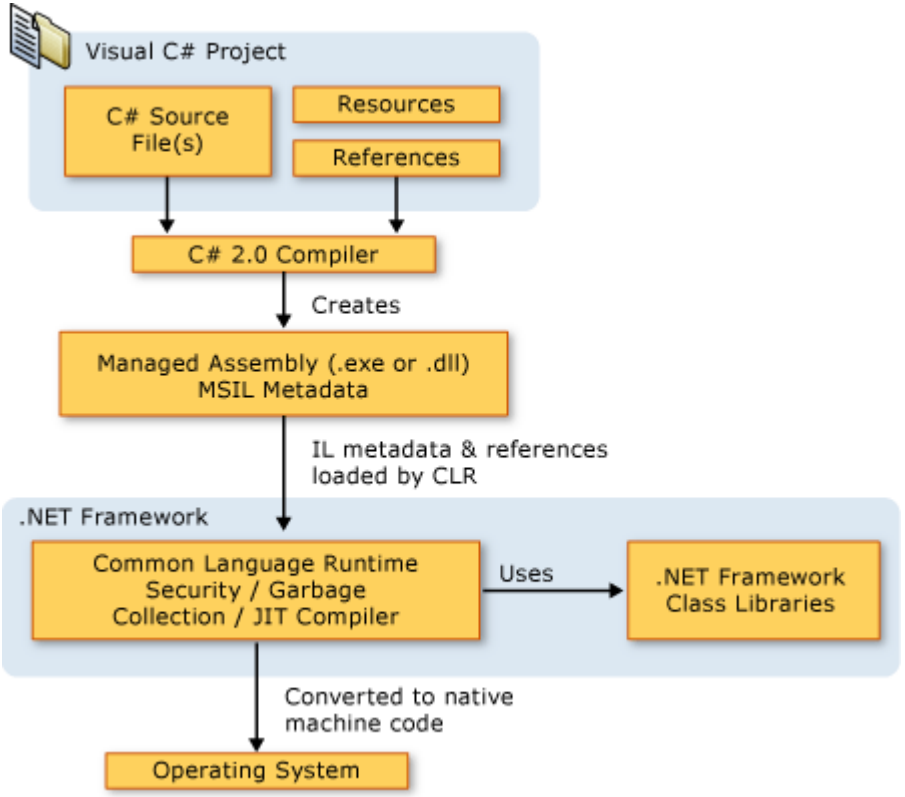
- C# - Caratteristiche
  - C# e il .NET Framework
  - Struttura di un programma C#
- I tipi
  - Enum, Classi, Struct
  - Interfacce
  - Delegate ed eventi
- Nozioni di base
  - Passaggio dei parametri
  - Array, Stringhe
  - Gestione degli errori
  - Classi generiche
  - Documentazione XML

C# - Caratteristiche generali

- C# è un linguaggio object-oriented della famiglia C/C++
  - Ogni variabile è un oggetto
  - Non solo Object-oriented ma Component-oriented
    - Un “componente” è un elemento software riutilizzabile e non dipendente dal contesto
    - Concetti di base: interfacce, proprietà, eventi
  - Principale ideatore: Anders Hejlsberg
    - Turbo Pascal e Delphi (Borland)
- Fortemente basato sul CLR di .NET e i tipi di dati definiti nel CLS
  - È stato appositamente progettato per trarre il massimo beneficio dal framework .NET
- Standard ISO: ISO/IEC 23270:2003 [http://www.iso.org]

C# e il .NET Framework

- Il compilatore C# produce codice intermedio MSIL
- Codice e risorse (es. bitmap) formano uno o più Assembly
- Assembly e manifest possono risiedere in .exe o .dll
- Il programma C# può utilizzare la Class Library che è messa a disposizione dal CLR
- Il CLR carica il codice MSIL e ne traduce le parti da eseguire in linguaggio macchina, compilandole con il JIT



Perché il C#?

- I linguaggi esistenti sono ricchi di funzionalità e potenti, era veramente necessario averne uno nuovo?
- Caratteristiche importanti erano sparse fra linguaggi diversi
  - Alcuni esempi:
    - Puntatori (C/C++)
    - Garbage collection (Java)
    - RAD: Rapid Application Development (Visual Basic)
- Perché C# in questo corso?
  - In un corso di questo tipo il linguaggio di programmazione "tradizionale" è il C/C++
    - Efficiente manipolazione di dati e immagini
    - Grande quantità di codice già esistente disponibile
  - Tuttavia la realizzazione di un programma Windows in C++ risulta piuttosto complicata, nonostante l'aiuto offerto dalla libreria MFC

Obiettivi del C#

- Fornire agli sviluppatori un unico linguaggio con:
  - Un insieme completo di potenti funzionalità
  - Una sintassi semplice e consistente
- Aumentare la produttività, eliminando problematiche tipiche
  - Type-safety
    - Ancora più rigido del C++ nelle conversioni fra tipi
    - Non sono consentite variabili non inizializzate
  - Garbage collection (non ci si deve preoccupare di liberare la memoria)
  - Gestione errori mediante eccezioni
  - Supporto per programmazione component-oriented
    - Proprietà, eventi, interfacce, attributi
  - Tipi unificati ed estensibili
    - Ogni cosa è un oggetto

C# - Caratteristiche

- Include caratteristiche di vari linguaggi
  - La sicurezza di Java
    - Completamente object-oriented, Garbage collection, Controllo dei limiti degli array a run-time, Gestione eccezioni
  - La semplicità di Visual Basic e Delphi
    - Proprietà ed eventi, Foreach, RAD (mediante Windows Forms di .NET)
  - La potenza ed espressività del C++
    - Enum, Overloading di operatori, Puntatori a funzione (delegate), Struct, passaggio dei parametri per riferimento o per valore, manipolazione diretta della memoria con puntatori (da usare con cautela...)
- Tutti i vantaggi del .NET Framework
  - Class Library con un vasto insieme di funzionalità già pronte
  - Funzionalità avanzate di comunicazione su Internet
  - Compilazione JIT

Hello World

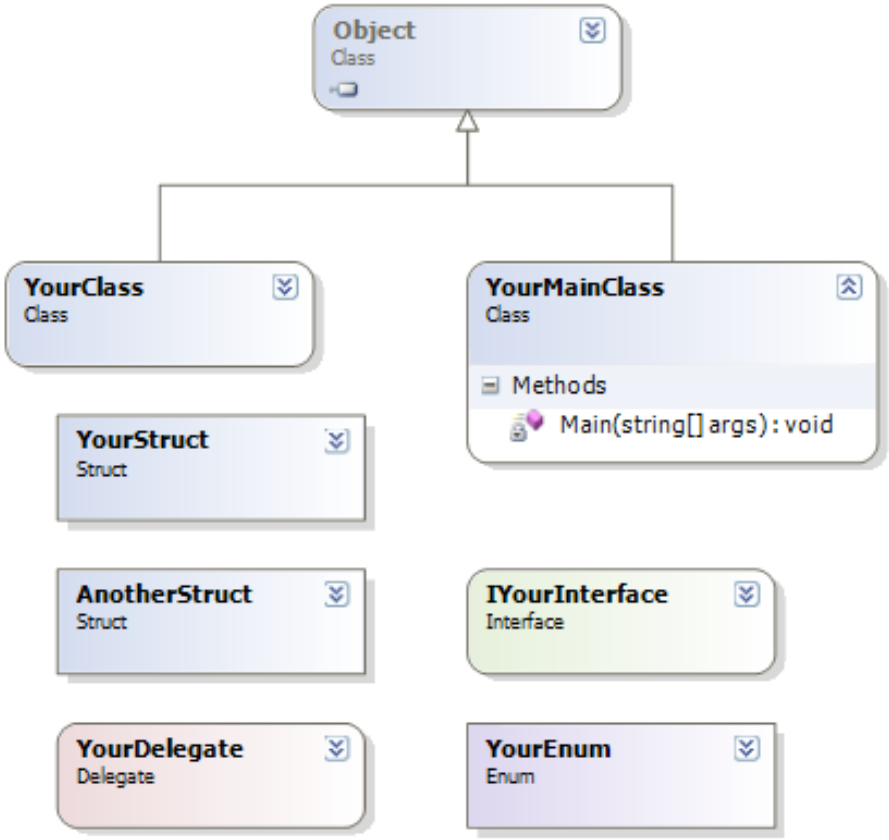
```
using System;
// Un programma "Hello World!" in C#
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Struttura di un programma C#

- Un programma C# è composto da uno o più file (*.cs)
- Ogni file contiene uno o più Namespace
- Namespace
  - Contiene tipi e/o altri Namespace
  - Definisce lo “scope” dei tipi che contiene
  - Permette di raggruppare il codice secondo criteri semantici
- Tipi
  - Class, Struct, Interface, Enum, Delegate
- Organizzazione del codice
  - Non vi sono file header (grazie agli Assembly), il codice è tutto scritto “in-line”
  - Non è necessario definire le classi e i metodi secondo un determinato ordine

Struttura di un programma C# (2)

```
using System;
namespace YourNamespace
{
    class YourClass
    {
    }
    struct YourStruct
    {
    }
    interface IYourInterface
    {
    }
    delegate int YourDelegate();
    enum YourEnum
    {
    }
    namespace YourNestedNamespace
    {
        struct AnotherStruct
        {
        }
    }
    class YourMainClass
    {
        static void Main(string[] args)
        {
            // Il programma inizia qui...
        }
    }
}
```

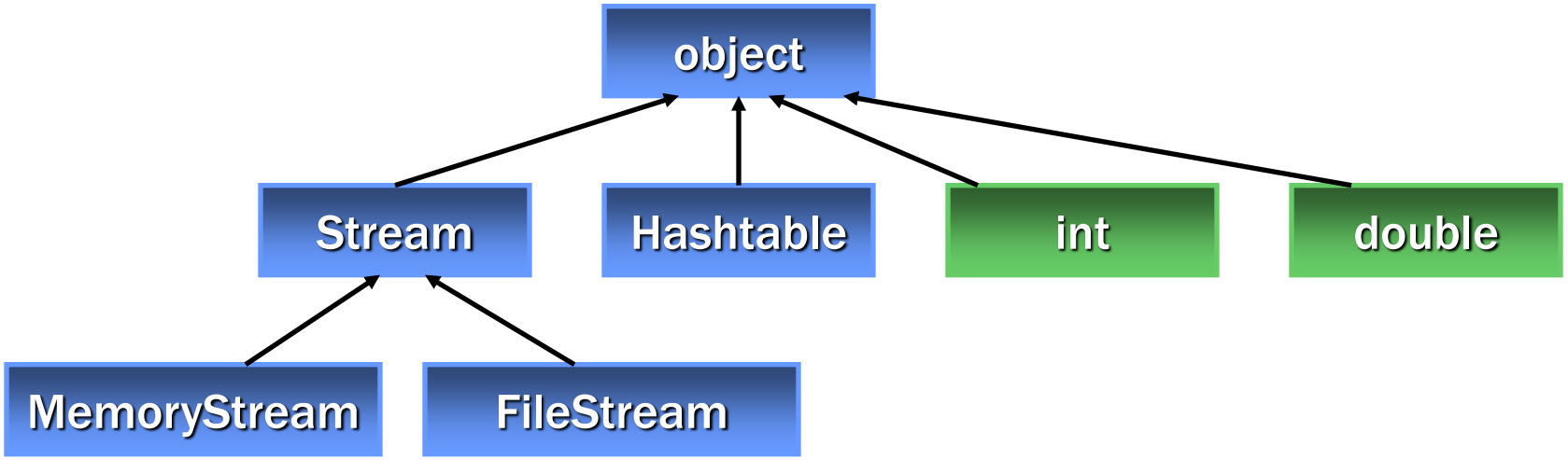


Sintassi e identificatori

- Sintassi
  - Simile al C/C++
  - Case-sensitive
  - Punto-e-virgola “;” per terminare istruzioni
  - Parentesi graffe { } per racchiudere blocchi di codice
  - Commenti in stile C++/Java
    - // Linea di commento
    - /* Inizio commento ... Fine Commento */
    - /// Documentazione XML
- Codifica: UNICODE
  - Sia per il codice sorgente che per le stringhe e i caratteri all’interno del programma
- Identificatori
  - Grammatica simile al C (ma codifica UNICODE; inoltre prefisso @ per poter utilizzare keyword come identificatori)

Tipi

- Ogni variabile deve avere un tipo
- Possibili tipi:
  - Tipi predefiniti (es. int, char)
  - Tipi definiti dall'utente (mediante class, struct)
- Sistema dei tipi unificato: tutto deriva dalla classe object



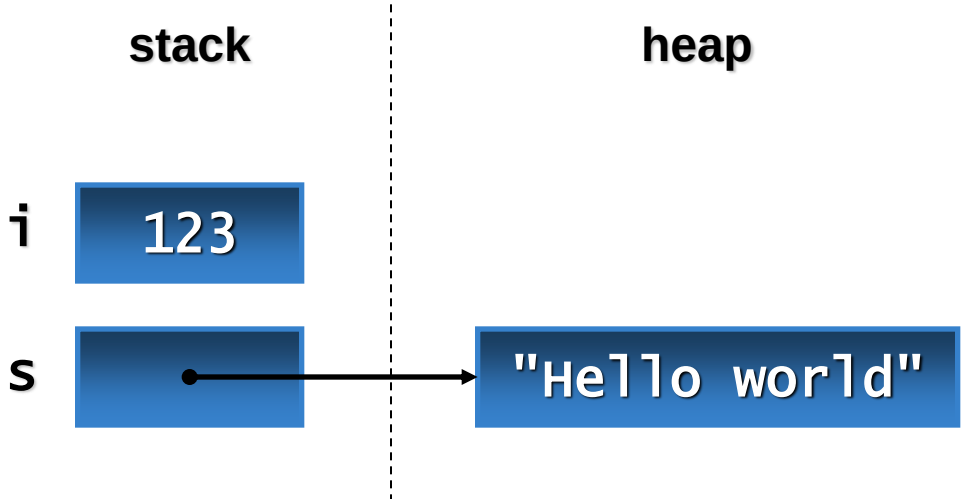
Tipi – Value Types e Reference Types

- Netta distinzione fra:
  - Value Types
    - Contengono direttamente il valore del dato
    - Non possono essere null (eccezione: Nullable Types)
    - L'assegnamento di una variabile a un'altra provoca la copia del valore contenuto nella variabile
  - Reference Types
    - Contengono un riferimento al valore del dato
    - Possono essere null
    - L'assegnamento di una variabile a un'altra provoca la copia del riferimento al valore e non del valore stesso (entrambe le variabili fanno quindi riferimento allo stesso dato)
- Tale distinzione è predefinita in base al tipo e non modificabile dal programmatore

Tipi – Value Types e Reference Types (2)

- Value type
  - Allocati nello stack (a meno che non siano membri di una classe)
  - Non necessitano di garbage collection
- Reference type
  - Fanno riferimento a oggetti allocati dinamicamente nello heap
  - Tali oggetti sono soggetti al garbage collection

```
...
int i = 123;
string s = "Hello world";
...
```



Tipi – Value Types e Reference Types (3)

Tipi		Descrizione
Value types	Simple types	Signed integral: sbyte, short, int, long
		Unsigned integral: byte, ushort, uint, ulong
		Unicode characters: char
		IEEE floating point: float, double
		High-precision decimal: decimal
		Boolean: bool
	Enum types	User-defined types of the form enum E {...}
	Struct types	User-defined types of the form struct S {...}
Reference types	Class types	Ultimate base class of all other types: object
		Unicode strings: string
		User-defined types of the form class C {...}
	Interface types	User-defined types of the form interface I {...}
	Array types	Single- and multi-dimensional, for example, int[] and int[,]
	Delegate types	User-defined types of the form delegate T D(...)

Tipi predefiniti

Categoria	V/R Type	Nome	Byte	Intervallo valori	Cifre significative
Intero con segno	Value	sbyte	1	-128 ... 127	-
		short	2	-32,768 ... 32,767	-
		int	4	-2^{31} ... $2^{31}-1$	-
		long	8	-2^{63} ... $2^{63}-1$	-
Intero senza segno	Value	byte	1	0 ... 255	-
		ushort	2	0 ... 65535	-
		uint	4	0 ... 4,294,967,295	-
		ulong	8	0 ... $2^{64}-1$	-
Carattere Unicode	Value	char	2	[U+0000,U+FFFF]	-
Floating point	Value	float	4	$\sim \pm 10^{-45}$ ... $\pm 10^{38}$	7
		double	8	$\sim \pm 10^{-324}$ ... $\pm 10^{308}$	15-16
		decimal	16	$\sim \pm 10^{-28}$ ... $\pm 10^{28}$	28-29
Booleano	Value	bool	1	true, false	-
Tipo generico	Reference	object	-	-	-
Stringa Unicode	Reference	string	-	-	-

Variabili tipizzate in modo implicito (C# 3.0)

- Le variabili locali possono avere un tipo *dedotto al momento dell'inizializzazione* anziché un tipo esplicito.
- Si utilizza la parola chiave var.
  - Tale parola chiave indica al compilatore di dedurre il tipo della variabile dall'espressione sul lato destro dell'istruzione di inizializzazione.
  - Attenzione: il c# è fortemente tipizzato, variabili di questo tipo hanno comunque un unico tipo che è determinato a compile-time, non a run-time.

```
...

int i = 123;
string s = "Hello world";
double d = 10.3;
UnaClasse r = new UnaClasse();
...
```

=

```
...

var i = 123;
var s = "Hello world";
var d = 10.3;
var r = new UnaClasse();
...
```

Enum

- Permette di definire un nuovo tipo che consiste in un insieme di valori costanti (ognuno con un nome)
  - Migliora la leggibilità del codice
  - Evita potenziali errori (gli enum non possono essere convertiti implicitamente in altri tipi di dati)
  - È possibile specificare il tipo di dato su cui l'enum è basato, determinandone l'occupazione di memoria (default: int)
  - Si possono indicare i corrispondenti valori numerici (default: 0, 1, 2, ...) e utilizzare operatori (es. OR bit a bit: '|')

```
enum UnEnum
{
PrimaCostante,
SecondaCostande,
UltimaCostante
}
```

```
enum MyColor : byte
{
    Black = 0,
    Red = 1,
    Green = 2,
    Blue = 4,
    Yellow = Red | Green,
    White = Red | Green | Blue,
}
```

Enum - Esempio

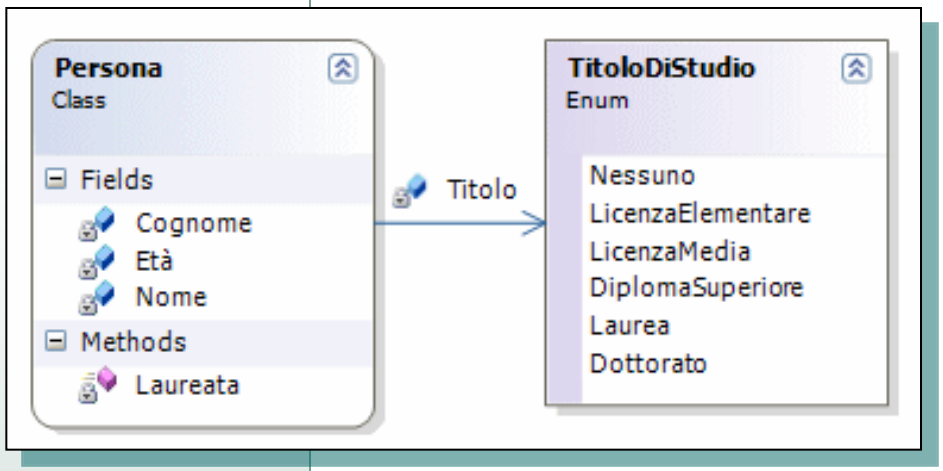
```
enum TitoloDiStudio
{
    Nessuno,LicenzaElementare,LicenzaMedia,
    DiplomaSuperiore,Laurea,Dottorato
}

static void Main()
{
    TitoloDiStudio titolo = TitoloDiStudio.DiplomaSuperiore;
    if (titolo==TitoloDiStudio.Nessuno)
    {
        Console.WriteLine("Nessun titolo di studio.");
    }
}
```

Classi

- Elementi fondamentali dei programmi C#
  - Una classe contiene sia codice che dati: definisce un nuovo tipo di dato sia nella sua struttura che nel suo “comportamento”
  - Gli elementi contenuti in una classe (membri) possono essere:
    - Campi: variabili che memorizzano i dati, possono essere inizializzati al momento della dichiarazione
    - Metodi: funzioni che implementano le azioni che la classe può compiere (ne definiscono il comportamento)
    - Proprietà: particolari metodi (set/get) accessibili come se fossero campi
    - Altri membri: eventi, operatori, indexer
    - Tipi annidati (interni alla classe): class, struct, delegate, enum, ...
- Unità semantiche atomiche
- Concetti importanti:
  - Oggetti, Ereditarietà e polimorfismo, Protezione dell’accesso, Interfacce

Classi - Esempio

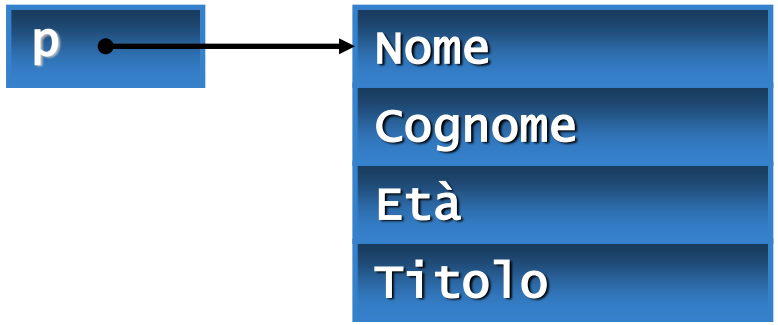
```
class Persona
{
  // Campi
  string Nome, Cognome;
  uint Età;
  TitoloDiStudio Titolo;

  // Metodi
  bool Laureata()
  {
    return Titolo >= TitoloDiStudio.Laurea;
  }
}
```



Classi e oggetti

- Due concetti differenti (benché a volte utilizzati come sinonimi)
  - Classe: è un tipo di dato, ma non il dato stesso
  - Oggetto: è una entità concreta basata su una certa classe (si dice che un oggetto è un'istanza di una classe)
  - La relazione fra oggetto e classe è analoga, ad esempio, a quella fra il numero "3" e il tipo "int"
- Creazione di un oggetto: keyword new
- Una variabile di una certa classe è un riferimento a un oggetto di tale classe (class è un Reference Type).
  - Valore predefinito null: non fa riferimento ad alcun oggetto

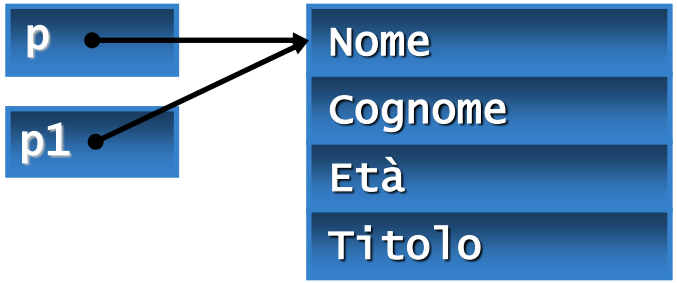
```
Persona p = new Persona();
Persona p1;
```



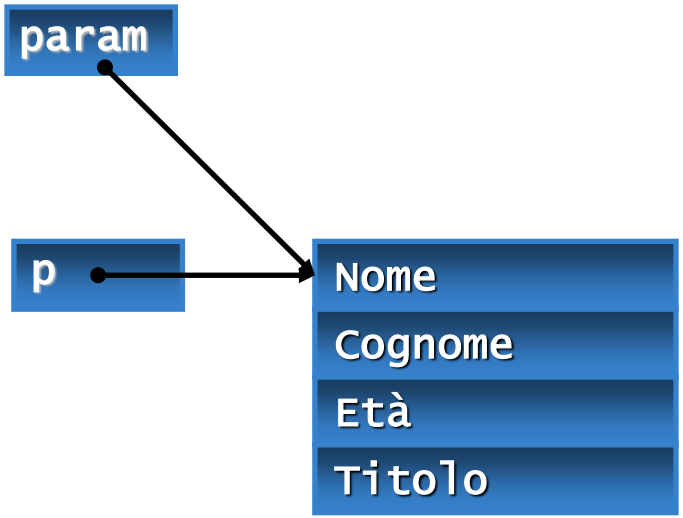
Classi e oggetti (2)

- La classe è un Reference Type – Alcune conseguenze:
  - L'operatore di assegnamento copia il riferimento, non l'oggetto
  - Un oggetto passato come parametro a una funzione è un riferimento allo stesso oggetto

```
Persona p = new Persona();
Persona p1 = p;
```




```
static void Func(Persona param)
{ ...
}
static void Main()
{   Persona p = new Persona();
    Func(p);
}
```



Protezione dell'accesso

- Ciascun membro di una classe può essere:
  - public – Accessibile da qualunque altra parte del codice
  - protected – Accessibile solo dalla classe stessa e da classi derivate
  - internal – Accessibile solo dall'Assembly corrente
  - protected internal – Accessibile solo dall'Assembly corrente o da classi derivate
  - private – Accessibile solo dalla classe stessa (default)

```
class Persona
{  // Campi
   private string Nome, Cognome;
   private uint Età;
   public TitoloDiStudio Titolo;
   ...
}
```

```
static void Main()
{
  Persona p = new Persona();
  p.Titolo =
    TitoloDiStudio.Dottorato;//OK
  p.Cognome = "Cappelli";
        // Errore di compilazione
}
```

Classi – Costruttore

- Metodo speciale (con lo stesso nome della classe) che viene chiamato alla creazione di un nuovo oggetto della classe
- Se la classe non contiene alcun costruttore, ne viene fornito uno di default dal compilatore (un costruttore senza alcun parametro)
- Una classe può avere più costruttori con parametri differenti: il programmatore può scegliere di volta in volta il più utile da chiamare in base al contesto

```
class Persona
{  ...
  public Persona(string nome, string cognome)
  {
      Nome = nome;  Cognome = cognome;
      Età = 0; Titolo = TitoloDiStudio.Nessuno;
  }
  public Persona(string nome, string cognome,
                 uint età, TitoloDiStudio titolo)
  {
      Nome = nome;  Cognome = cognome;
      Età = età;  Titolo = titolo;
  }
...
```

```
static void Main()
{
    Persona padre = new Persona(
          "Luca","Bianchi",20,
          TitoloDiStudio.Laurea
          );

    Persona figlio = new Persona(
          "Sara", "Bianchi“
          );
}
```

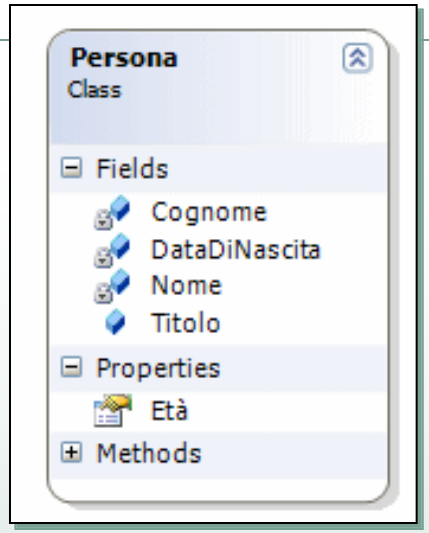
Proprietà

- Coniugano la semplicità dei campi alla flessibilità dei metodi
- Vi si accede come campi (es. if (Persona.Età>=18) ...) ma sono in realtà dei metodi:
  - get : chiamato quando viene letto il valore della proprietà
  - set : chiamato quando viene assegnato un nuovo valore alla proprietà
- Utili per:
  - Implementare campi read-only (definendo solo il metodo get)
  - Validazione e/o altri aggiornamenti al momento dell'assegnamento
  - Valori derivati e/o composti
  - Esporre valori attraverso un'interfaccia

```
class UnaClasse
{
    private int campo = 0;
    public int UnaProprietà
    {
        get { return campo; }
        set { campo = value; }
    }
}
```

Proprietà – Esempio

```
class Persona
{
  private string Nome, Cognome;
  private DateTime DataDiNascita;
  public TitoloDiStudio Titolo;

  public int Età  // Proprietà di tipo int (read-only)
  {
    get
    {
      DateTime now = DateTime.Now;
      DateTime dn = DataDiNascita;
      int età = now.Year - dn.Year;
      if (now.Month<dn.Month || (now.Month==dn.Month && now.Day<dn.Day))
        età--;
      return età;
    }
  }

...
```




```
static void Main()
{
  Persona padre = new Persona("Luca","Bianchi",...);
  Console.WriteLine(padre.Età);
}
```

Proprietà automatiche (C# 3.0)

- Proprietà auto-implementate
  - Permettono di rendere la dichiarazione delle proprietà più semplice e concisa.
  - Utili per proprietà che semplicemente permettono l'accesso a un membro privato, senza implementare logiche specifiche nei metodi di accesso (get/set).

```
class Persona
{
  public string Nome { get; set; }
  public string Cognome { get; set; }
  public DateTime DataDiNascita { get; private set; }
}
```

=

```
class Persona
{
    private string nome;
    private string cognome;
    private DateTime dataDiNascita;
    public string Nome
    {
       get { return nome; }
       set { nome = value; }
    }
    public string Cognome
    {
       get { return cognome; }
       set { cognome = value; }
    }
    public DateTime DataDiNascita
    {
       get { return dataDiNascita; }
    }
}
```

Inizializzatori (C# 3.0)

- Object initializers
  - Permettono di assegnare un valore a proprietà e campi pubblici di una classe al momento della creazione di una sua istanza, senza che vi sia un costruttore che esplicitamente accetta tali parametri.

```
class Persona
{
  public string Nome { get; set; }
  public string Cognome { get; set; }
  public DateTime DataDiNascita { get; private set; }
}
```

```
...
var p1 = new Persona { Nome = "Pluto" };
var p2 = new Persona { Nome = "Mario", Cognome="Rossi" };
...
```

=

```
...
var p1 = new Persona();
p1.Nome = "Pluto";
var p2 = new Persona();
p2.Nome = "Mario";
p2.Cognome = "Rossi";
...
```

Membri statici

- La keyword static permette di dichiarare membri di una classe che appartengono al tipo di dati stesso: ve ne è una sola istanza a prescindere dal numero di oggetti creati e possono essere utilizzati anche se non è stato creato alcun oggetto di tale classe
  - Possono essere utili per memorizzare informazioni (o, nel caso di metodi, fornire funzionalità) che sono legate dal punto di vista semantico alla classe nel suo insieme e non a una sua determinata istanza

```
class Persona
{
  ...
  private static int NumIstanze = 0; // campo statico
  public static int NumeroPersone()  // metodo statico
  { return NumIstanze; }
  public Persona(string nome, string cognome) // costruttore
  {
    NumIstanze++; // Ogni costruttore incrementa il contatore
    Nome = nome;
    ...
  }
  ...
}
```

```
...
Console.WriteLine(Persona.NumeroPersone());
...
```

Classi – parola chiave «this»

- La keyword this fornisce un riferimento all'istanza corrente di una classe
- È principalmente utilizzata per:
  - Fare riferimento a campi della classe quando vi sono variabili o parametri con lo stesso nome
  - Passare l'oggetto (istanza) corrente come parametro a un metodo
- Membri static non possono ovviamente utilizzare la keyword this (così come non possono chiamare metodi non static e in generale fare riferimento a campi non static).

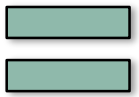
```
class B
{
    public void func(A obj)
    { ... }
}

class A
{
    int a;
    public A(int a)
    {
        this.a = a;
        B b = new B();
        b.func(this);
    }
}
```

Metodi di estensione (C# 3.0)

- Gli extension method consentono di aggiungere metodi ai tipi esistenti, senza creare un nuovo tipo derivato.
- Parola chiave «this»:
  - Precede la dichiarazione del primo parametro in un metodo statico.
  - Tale metodo statico può essere chiamato come se fosse un metodo (non statico) del tipo del primo parametro.

```
static class Estensioni
{
  public static int Conta(this string testo, char carattere)
  {
    int n = 0;
    foreach (var c in testo)
    {
      if (c == carattere) n++;
    }
    return n;
  }
}
```

```
var t = "Hello";
var n1 = t.Conta('l');
var n2 = "TESTO".Conta('T');
```

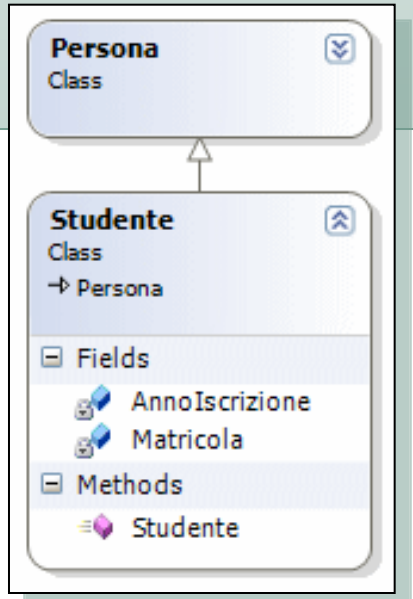


```
var t = "Hello";
var n1 = Estensioni.Conta(t,'l');
var n2 = Estensioni.Conta("TESTO",'T');
```

Ereditarietà

- Una classe (classe derivata) può ereditare da un'altra classe (classe base)
- La classe derivata eredita dati e comportamento della classe padre, ai quali può aggiungere altri dati e funzionalità
- La keyword base permette a una classe derivata di fare riferimento alla classe base
- Il costruttore di una classe derivata tipicamente richiama un costruttore della classe base

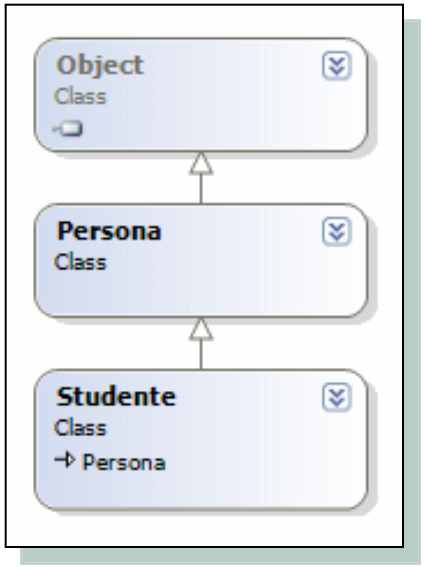
```
class Studente : Persona
{
  // Campi
  private String Matricola;
  private int AnnoIscrizione;

  // Costruttore
  public Studente(String n, String c)
    : base(n, c)
  {
    Matricola = CreaNuovaMatricola();
    AnnoIscrizione = DateTime.Now.Year;
  }
  ...
}
```



Polimorfismo

- L'ereditarietà fa sì che un oggetto possa appartenere a più di un tipo (la sua classe e la classe base)
- Conversione:
  - Type cast (sintassi simile al C)
    - Se la conversione non può essere eseguita → errore (eccezione)
  - Operatore as
    - Se la conversione non può essere eseguita il risultato è null
- Controllo:
  - Operatore is
    - Permette di controllare se un oggetto appartiene a un determinato tipo

```
Persona A = new Persona("a", "A");
Studente B = new Studente("b", "B");
Persona C = B; // Corretto: non richiede cast
Studente D = (Studente)C; // Corretto ma richiede cast
Studente E = (Studente)A; // Genera errore a run-time
Studente F = C as Studente; // Corretto: F=C
Studente G = A as Studente; // Corretto: G=null
bool b1 = (A is Persona) && (B is Studente); // true
bool b2 = F is Studente; // true
bool b3 = A is Studente; // false
bool b4 = G is Studente; // false
bool b5 = B is Object; // true
```



Polimorfismo (2)

- Una classe derivata può ridefinire membri della classe base utilizzando la keyword new
- Una classe derivata può ridefinire *completamente* i membri che la classe base ha dichiarato virtuali (virtual)
  - Metodi e proprietà possono essere virtual (non i campi)
  - La dichiarazione del metodo che ridefinisce un metodo virtuale va accompagnata dalla keyword override
  - Un metodo virtuale ridefinito, viene chiamato a prescindere dal tipo della variabile su cui è invocato
- È possibile dichiarare una classe come abstract
  - Classi astratte non possono essere istanziate (non si può creare un oggetto di tali classi): sono tipicamente utilizzate per fornire funzionalità comuni a una serie di classi derivate
  - Contengono tipicamente uno o più metodi abstract (metodi virtual senza implementazione)

Polimorfismo – Esempi

```
class Persona
{ ...
  public void StampaDesc()
  {
      Console.WriteLine(
        "{0} {1}, anni {2}",
        Nome, Cognome, Età);
  }
}
class Studente : Persona
{ ...
  public new void StampaDesc()
  {
      base.StampaDesc();
      Console.WriteLine(
        "Mat: {0}, Isc: {1}",
        Matricola,AnnoIscrizione);
  }
}
```

```
Persona A = new Persona("a", "A");
Studente B = new Studente("b", "B");
Persona C = B;
A.StampaDesc(); // Persona.StampaDesc()
B.StampaDesc(); // Studente.StampaDesc()
C.StampaDesc(); // Persona.StampaDesc()
```

```
class Persona
{ ...
  public virtual void StampaDesc()
  {
      Console.WriteLine(
        "{0} {1}, anni {2}",
        Nome, Cognome, Età);
  }
}
class Studente : Persona
{ ...
  public override void StampaDesc()
  {
      base.StampaDesc();
      Console.WriteLine(
        "Mat: {0}, Isc: {1}",
        Matricola,AnnoIscrizione);
  }
}
```

```
Persona A = new Persona("a", "A");
Studente B = new Studente("b", "B");
Persona C = B;
A.StampaDesc(); // Persona.StampaDesc()
B.StampaDesc(); // Studente.StampaDesc()
C.StampaDesc(); // Studente.StampaDesc()
```

Struct

- Strutture – simili alle classi ma con importanti differenze
  - Value type (class è un Reference type)
  - Non hanno ereditarietà
  - Possono contenere metodi
  - Possono avere costruttori (ma devono necessariamente avere dei parametri, non si può specificare un costruttore di default senza parametri)
  - I campi non possono essere inizializzati al momento della dichiarazione (eccetto quelli static)
- Ideali per oggetti "leggeri"
  - Sono allocate nello stack (a differenza degli oggetti che sono sempre allocati nello heap)
    - Utilizzo più efficiente della memoria (non serve garbage collection)
  - int, float, double, ..., sono tutte struct!

```
struct Vector
{
    private double vx, vy;
    public Vector(double vx,double vy)
    {
        this.vx = vx;
        this.vy = vy;
    }
    public double CalculateNorm()
    {
        return Math.Sqrt(vx*vx + vy*vy);
    }
}
```

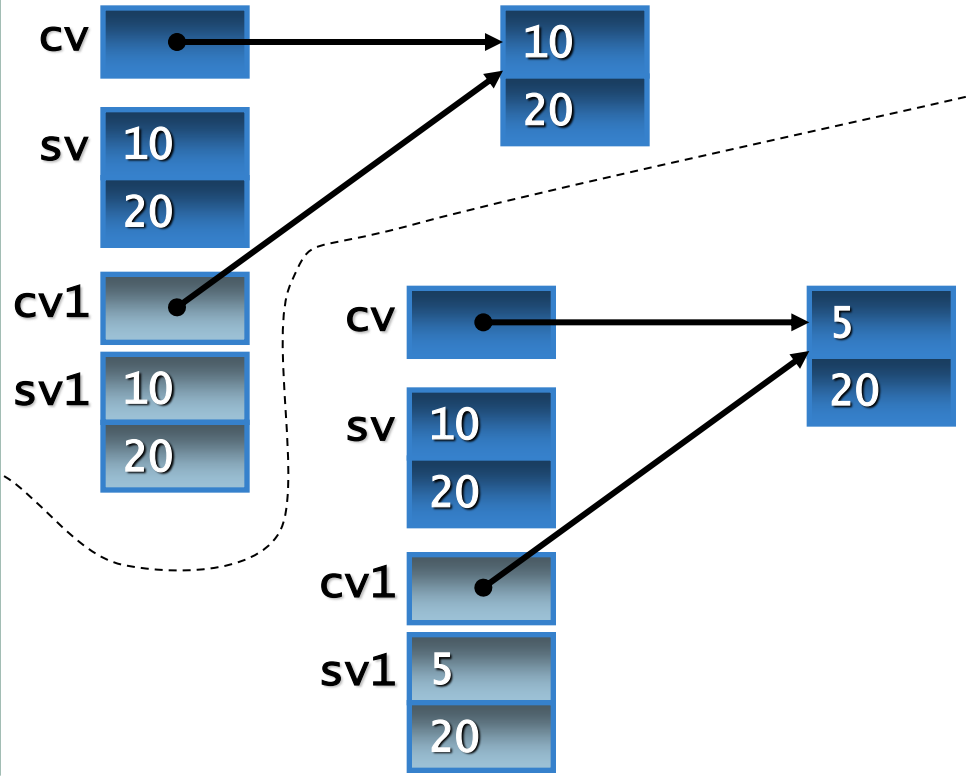
```
Vector P; //Alloca la struttura senza
          //new: i campi sono
          //inizializzati ai valori
          //di default

Vector P1 = new Vector(2,10);
          // Alloca la struttura chiamando
          // un costruttore con parametri

double n = P1.CalculateNorm();
```

Struct e classi

```
class CVector
{ public double vx, vy; ... }

struct SVector
{ public double vx, vy; ... }

CVector cv = new CVector(10,20);

SVector sv = new SVector(10,20);

CVector cv1 = cv;

SVector sv1 = sv;


cv1.vx = 5;
sv1.vx = 5;

Console.WriteLine(sv.vx); // 10
Console.WriteLine(cv.vx); // 5!
```

Class: Reference Type

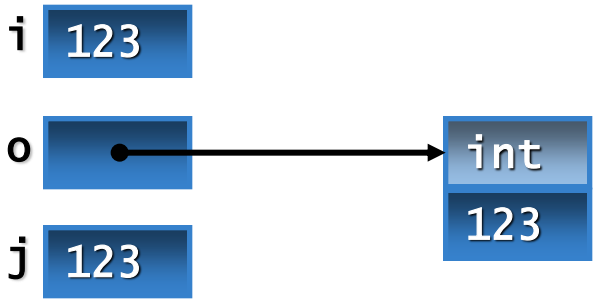
Struct: Value Type



Boxing e Unboxing

- Sistema dei tipi unificato
  - Tutti i tipi derivano implicitamente da object
  - Ma come è possibile convertire un Value Type in un object (Reference Type)?
- Conversione Boxing: Value Type → object
  - Alloca un apposito oggetto della classe object e vi copia il valore dentro
- Conversione Unboxing: object → Value Type
  - Controlla il tipo del valore contenuto e, se possibile, copia il valore dall'oggetto nella variabile Value Type

```
int i = 123;

object o = i;   // Boxing

int j = (int)o; // Unboxing
```



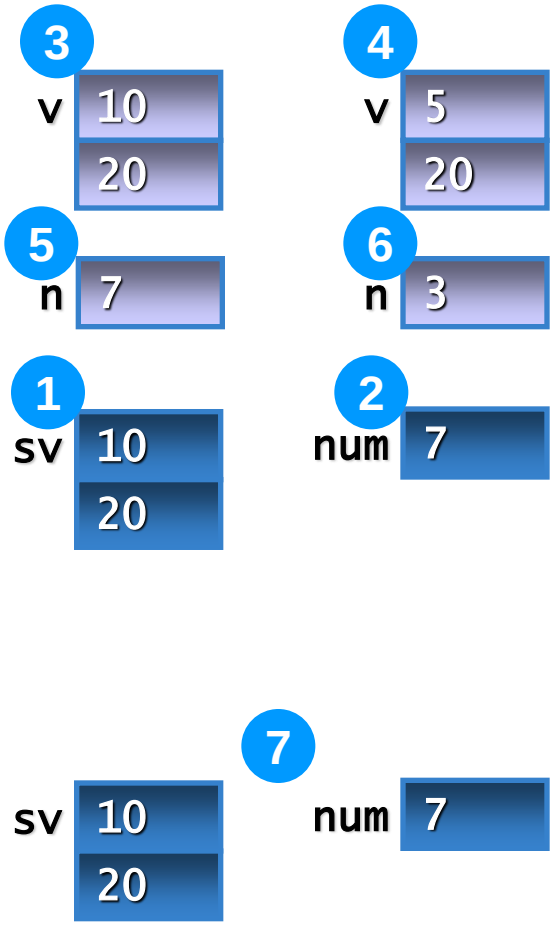
Passaggio dei parametri

- In C# si possono passare i parametri:
  - per valore (modalità predefinita) – viene creata una copia della variabile passata dal chiamante
  - per riferimento (keyword ref) – è possibile modificare direttamente la variabile passata dal chiamante
    - Un caso particolare del passaggio per riferimento si ha utilizzando la keyword out invece di ref: in tale caso non è necessario che il parametro sia inizializzato prima della chiamata (utile per funzioni che hanno più di un valore di output)
- Attenzione:
  - La modalità di passaggio dei parametri (per valore o per riferimento) non va confusa con il tipo di variabile (Value Type o Reference Type)
  - È possibile passare Value Types sia 1) per valore che 2) per riferimento e passare Reference Types sia 3) per valore che 4) per riferimento

Passaggio dei parametri – Esempi (1)

- Value Types passati per valore

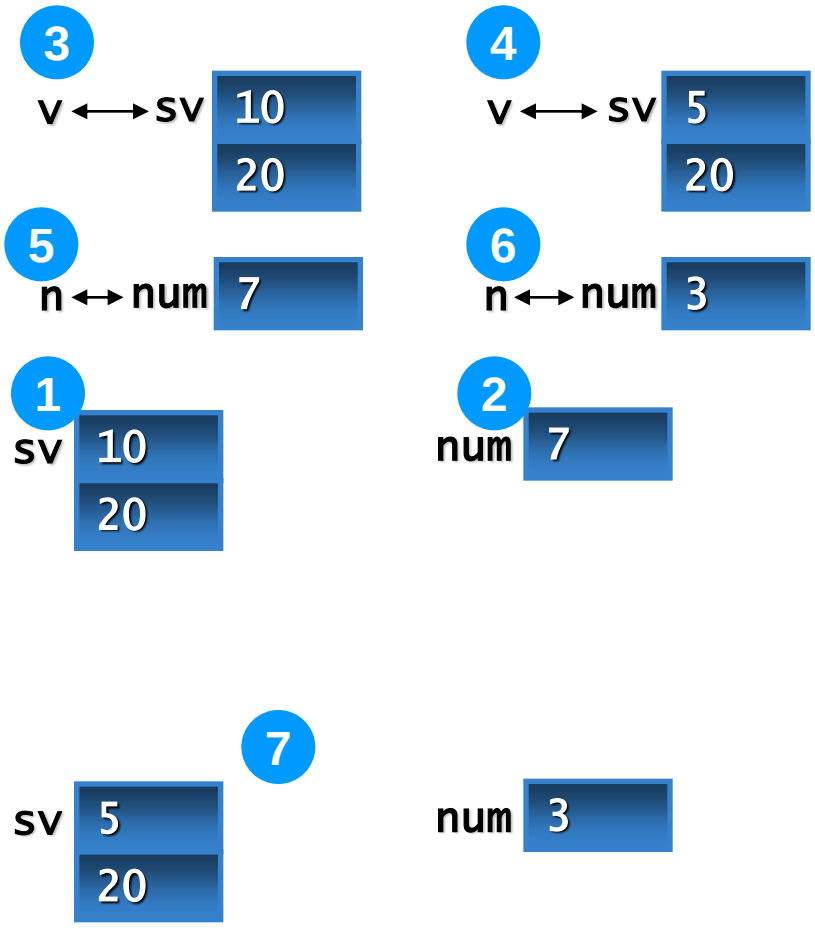
```
struct SVector
{ public double vx, vy; ... }
void ModifyVect(SVector v)
{ v.vx = 5; } 4
void ModifyInt(int n)
{ n = 3; } 6
...
SVector sv = new SVector(10, 20); 1
int num = 7; 2

ModifyVect(sv); 3
ModifyInt(num); 5

Console.WriteLine(sv.vx); // 10
Console.WriteLine(num); // 7 7
```



Passaggio dei parametri – Esempi (2)

- Value Types passati per riferimento

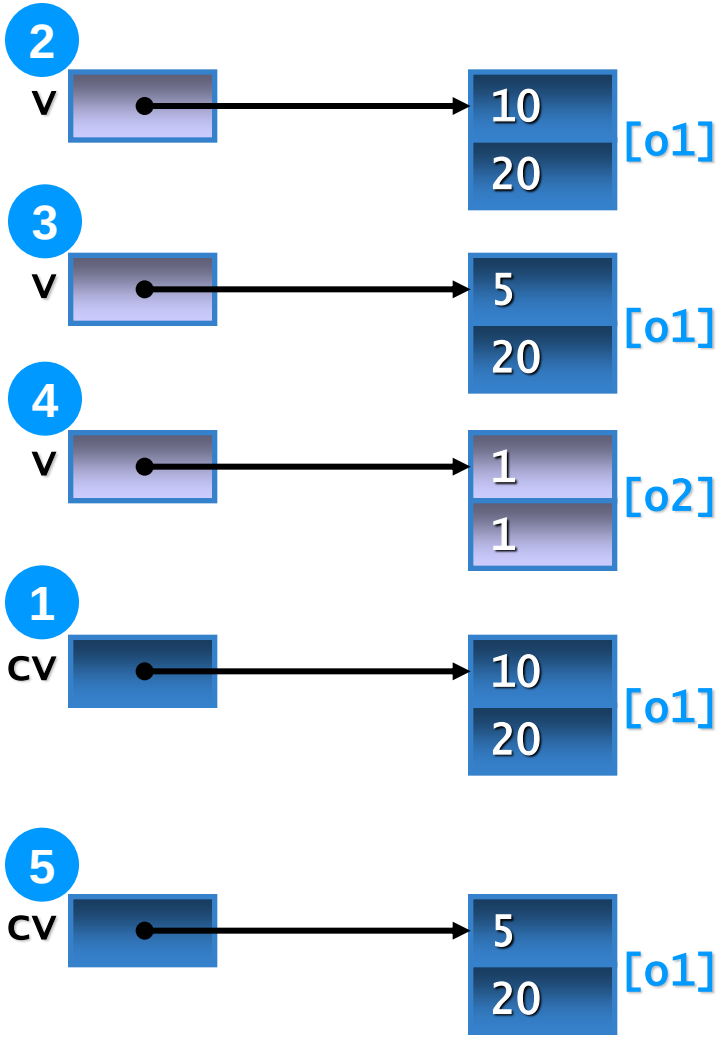
```
struct SVector
{ public double vx, vy; ... }
void ModifyVect(ref SVector v)
{ v.vx = 5; } 4
void ModifyInt(ref int n)
{ n = 3; } 6
...
SVector sv = new SVector(10, 20); 1
int num = 7; 2

ModifyVect(ref sv); 3
ModifyInt(ref num); 5

Console.WriteLine(sv.vx); // 5
Console.WriteLine(num); // 3 7
```



Passaggio dei parametri – Esempi (3)

- Reference Types passati per valore

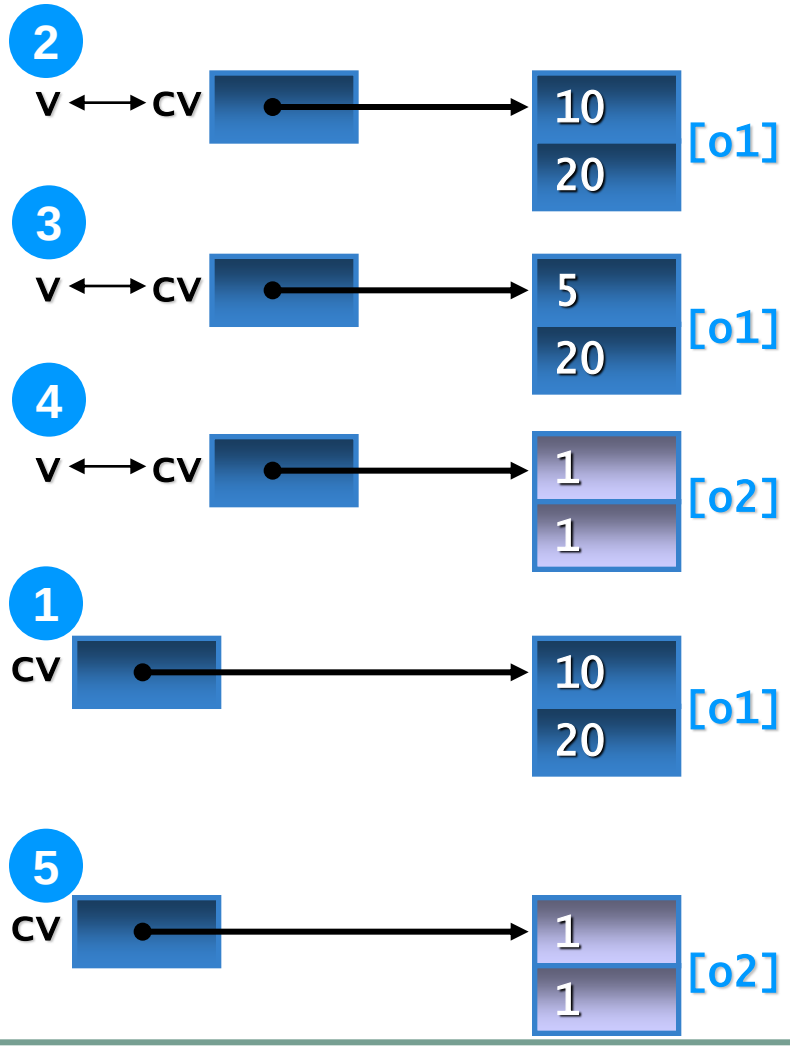
```
class CVector
{ public double vx,vy; ... }

void ModifyVect(CVector v)
{
  v.vx = 5;   3
  v = new CVector(1, 1);   4
}

...
CVector cv = new CVector(10, 20);   1

ModifyVect(cv);   2

Console.WriteLine("{0},{1}",   5
                  cv.vx,cv.vy);
```



Passaggio dei parametri – Esempi (4)

- Reference Types passati per riferimento

```
class CVector
{ public double vx,vy; ... }

void ModifyVect(ref CVector v)
{
  v.vx = 5;
  v = new CVector(1, 1);
}

...
CVector cv = new CVector(10, 20);

ModifyVect(ref cv);

Console.WriteLine("{0},{1}",
                  cv.vx,cv.vy);
```



Array

- Reference Type derivato dalla classe abstract Array
  - Gli array possono essere:
    - Mono-dimensionali
    - Multi-dimensionali
    - Jagged (array di array)
  - Possono contenere qualsiasi tipo (value type o reference type)
  - Gli indici partono da 0: [0..n-1]
  - Valori di default degli elementi:
    - Numerici: 0
    - Reference types: null
- Un array è un oggetto:
  - Proprietà `Length`
  - Metodo `Clone()`
  - `...`

```
// Array monodimensionale: valori a 0
int[] vet1 = new int[30];

// Inizializza gli elementi con un ciclo
for (int i = 0; i < vet1.Length; i++)
    vet1[i] = i;
```

Array – Esempi

```
// Array monodimensionale: valori a 0
int[] vet1 = new int[9];
// Array monodimensionale: inizializzazione
int[] vet2 = new int[5] {1,2,3,6,12};
// Inizializzazione: sintassi alternativa
int[] vet3 = {1,2,3,6,12};  // Equivale a new..

// Array bidimensionale (matrice)
double[,] mat1 = new double[3, 4];
// Inizializzazione array bidimensionale (per righe)
double[,] mat2 = { {1,2,3}, {4,5,6}, {7,8,9} };
double val = mat2[2,1]; // 8

// Array "jagged"
byte[][] jag = new byte[3][];
// Crea i sotto-array dell'array jagged
for (int i = 0; i < jag.Length; i++)
    jag[i] = new byte[3 + i * 2];
jag[2][3] = 23;
```

0	0	0	0	0	0	0	0	0

1	2	3	6	12

1	2	3	6	12

0	0	0	0
0	0	0	0
0	0	0	0

1	2	3
4	5	6
7	8	9

0	0	0				
0	0	0	0	0		
0	0	0	23	0	0	0

Array – Esempi (2)

```
int[] vet1 = {5,2,3,4,1};
int[] vet2 = vet1;  // N.B. è un reference type: non crea una copia

// Alcune proprietà della classe Array
float[] v = new float[5];
int nv = v.Length; // 5
double[,,] vet3D = new double[20, 30, 40];
int r = vet3D.Rank; // 3
int tot = vet3D.Length; // 24000 (20*30*40)

// Alcuni metodi della classe Array
int n1 = vet3D.GetLength(0); // 20
int n2 = vet3D.GetLength(1); // 30
int[] vet3 = (int[])vet1.Clone(); // Crea una copia
vet3 = new int[3] { 10, 20, 30 };
vet3.CopyTo(vet1, 2); // Copia elementi: vet1={5,2,10,20,30};

// Alcuni utili metodi static della classe Array
Array.Sort(vet1);   // {2,5,10,20,30}
int index = Array.BinarySearch(vet1, 5); // index=1
Array.Reverse(vet1); // {30,20,10,5,2}
```

Indexer

- Un indexer permette di accedere a istanze di classi o struct come se fossero array
  - Si utilizza la keyword this nella dichiarazione
  - Dichiarazione simile a quella delle proprietà (get/set)
  - È possibile utilizzare indexer mono- o multi-dimensionali (esattamente come per gli array)
  - Gli indici non devono necessariamente essere interi

```
class MyArray
{
  private int[] arr = new int[100];
  public int this[int index]
  {
    get
    {
     return (index<0 || index>=100) ?
                                0 : arr[index];
    }
    set
    {
      if (index >= 0 && index < 100)
         arr[index] = value;
    }
  }
}
...
MyArray test = new MyArray();
test[3] = 256;
test[5] = 1024;
for (int i = 0; i <= 10; i++)
  Console.WriteLine("Elemento {0} = {1}",
                                  i, test[i]);
```

Stringhe

- Tipo predefinito (Reference Type)
  - Sequenza di caratteri Unicode
  - Operatori != e == ridefiniti: confrontano l'effettivo contenuto delle stringhe (altrimenti essendo Reference Type avrebbero confrontato solo i riferimenti)
  - Oggetto non-modificabile (una volta inizializzata, una stringa non può essere più modificata)
    - Tutti i metodi che effettuano modifiche alla stringa in realtà creano un nuovo oggetto; se sono necessarie modifiche frequenti a una sequenza di caratteri, è disponibile una classe apposita nella Class Library: StringBuilder
- Funzionalità
  - Ampia scelta di metodi (es. IndexOf()) e proprietà (es. Length)
  - Classi per svariate operazioni (es. Espressioni regolari) nella Class Library (namespace System.Text)
- Costanti di tipo stringa
  - Sequenze di escape (simili al C): "...\t...\n"
  - Simbolo @ per ignorare escape: @"c:\temp\file.txt" ("c:\\temp\\file.txt")

Istruzioni

- Controllo di flusso
  - if, else, switch, case: analoghe al C
  - break, continue, default, goto, return: simili al C
  - yield: utilizzato negli iteratori
- Cicli
  - do, for, while: analoghe al C
  - foreach, in: permette di eseguire un'operazione su tutti gli elementi di un array o di altre strutture dati
- Gestione delle eccezioni
  - throw, try-catch-finally
- Altre istruzioni
  - checked, unchecked: per abilitare/disabilitare il controllo dell'overflow
  - fixed: Utilizzato in codice unsafe
  - lock: Per definire sezioni critiche (mutua esclusione fra thread)

Operatori (in ordine di precedenza)

Primari	x.y, f(x), a[x], x++, x--, new, typeof, checked, unchecked, default(T), delegate, ->
Unari	+, -, !, ~, ++x, --x, (Tipo)x
Moltiplicativi	*, /, %
Additivi	+, -
Shift dei bit	<<, >>
Relazionali e controllo dei tipi	<, >, <=, >=, is, as
Uguaglianza	==, !=
Logici (sia bit a bit che booleani)	&, ^, | *(in ordine di precedenza)*
And/Or booleani	&&, || *(in ordine di precedenza)*
Null-coalescing	??
Condizionale	?:
Assegnamento	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
Operatore Lambda (C# 3.0)	=>

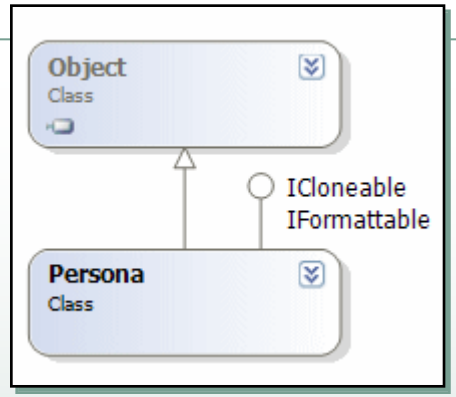
Interfacce

- Una interfaccia
  - Contiene solo la dichiarazione (non l'implementazione) di metodi e proprietà
  - Definisce struttura e semantica di specifiche funzionalità
  - Un'interfaccia può ereditare da una o più interfacce
  - Una classe può ereditare da una sola classe base, ma può implementare una o più interfacce
  - Convenzione: i nomi delle interfacce iniziano con '`I`'
- Esempi di interfacce presenti nella Class Library:

```
public interface ICloneable
{
   object Clone();
}
```

```
public interface IFormattable
{
  string ToString(string format,
         IFormatProvider formatProvider);
}
```

Interfacce – Esempio




```
class Persona : ICloneable, IFormattable
{
  private string Nome, Cognome;
  private DateTime DataDiNascita;
  public object Clone() // Implementa ICloneable
  { // Costruisce una copia
    return new Persona(Nome, Cognome, DataDiNascita);
  }
  public string ToString(string format, IFormatProvider formatProvider)
  {  // Implementa IFormattable
    if (format==null) // Nome e cognome
      return String.Format("{0} {1}",Nome,Cognome);
    else if (format == "ALL") // Nome, cognome e data di nascita
      return String.Format("{0} {1} nato il {2:d}",Nome,Cognome,DataDiNascita);
    else throw new FormatException("Formato sconosciuto: " + format);
  }
  ...
}
```

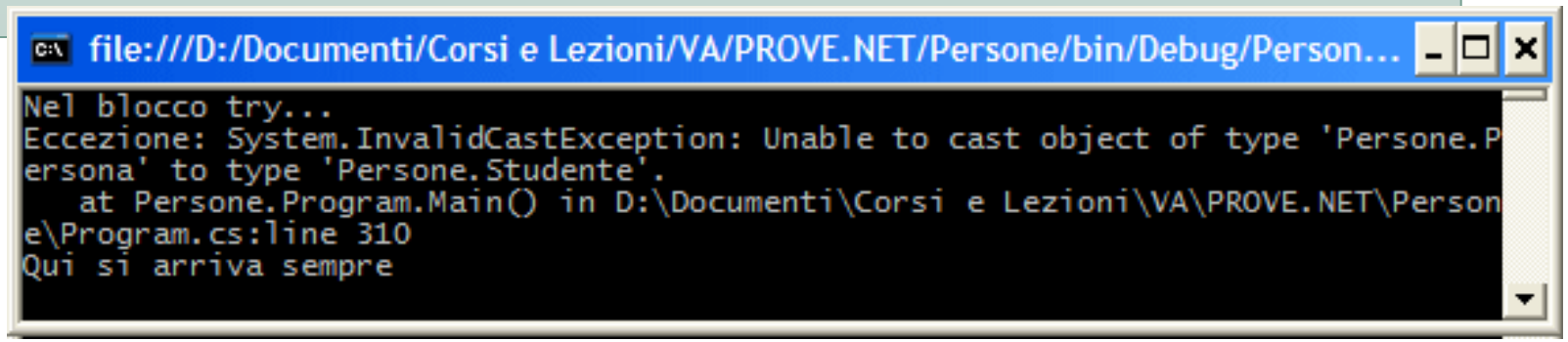
```
Persona A = new Persona("Carlo","Rossi",new DateTime(1980,10,20));
Persona B = new Persona("Luca","Bianchi",new DateTime(1988,11,30));
Persona A1 = (Persona)A.Clone(); // Utilizza ICloneable
Console.WriteLine("{0} - {1:ALL}", A, B); // Utilizza IFormattable
```

Gestione degli errori

- Eccezione
  - Situazione anomala che potrebbe avvenire durante l'esecuzione del programma
- try {...} catch(...) {...} – finally {...}
  - try: delimita un blocco di codice in cui "intercettare" eventuali eccezioni
  - catch(TipoEccezione): permette di "intercettare" e gestire un particolare tipo di eccezione
  - catch senza argomento: permette di gestire qualsiasi eccezione avvenga nel blocco try
  - finally: blocco di codice eseguito sempre e comunque al termine del try-catch, in caso di eccezione o meno

Gestione degli errori – Esempio

```
Persona A = new Persona("a", "A", ... );
try
{ Console.WriteLine("Nel blocco try...");
  Studente S = (Studente)A; // Eccezione: invalid cast
  Console.WriteLine("Io non vengo mai stampato a video");
}
catch (InvalidCastException e)
{ Console.WriteLine("Eccezione: {0}", e);
}
finally
{ Console.WriteLine("Qui si arriva sempre");
  // Tipicamente utilizzato per operazioni che vanno
  // svolte in ogni caso (es. chiusura di un file)
}
```



Delegate

- Simili ai puntatori a funzione del C/C++, ma
  - Orientati agli oggetti
  - Type-safe
- Caratteristiche principali
  - Permettono di passare un metodo come parametro o di assegnarlo a una variabile
  - Una volta che a una variabile di tipo delegate è stato assegnato un metodo, si comporta esattamente come tale metodo
  - Più metodi possono essere assegnati allo stesso delegate: quando il delegate è chiamato, sono eseguiti in sequenza
  - Sono alla base degli eventi

```
delegate void StatoAvanz(int perc);

class Esecutore
{
 public void Esegui(StatoAvanz callBack)
 {
   for (int i = 0; i < 1000; i++)
   { ...
     if (i%100==0)
       callBack(i / 10);
     ...
   }
 }
}
...
void StampaStato(int perc)
{ Console.Write("...{0} ", perc); }

void IgnoraStato(int perc) { }

...
Esecutore E = new Esecutore();
E.Esegui(StampaStato);
E.Esegui(IgnoraStato);
```

Delegate con metodi anonimi

- Permettono di passare direttamente un “blocco di codice” a un parametro di tipo delegate
  - Eliminano la necessità di dichiarare un metodo separato per poi poterlo passare al delegate
  - La keyword delegate sostituisce il nome del metodo (che è appunto “anonimo”) ed è seguita dalla dichiarazione degli eventuali parametri del metodo
- L’esempio a fianco equivale a quello del lucido precedente, ma utilizza metodi anonimi

```
delegate void StatoAvanz(int perc);

class Esecutore
{
 public void Esegui(StatoAvanz callBack)
 {
   for (int i = 0; i < 1000; i++)
   { ...
     if (i%100==0)
       callBack(i / 10);
     ...
   }
 }
}
...
Esecutore E = new Esecutore();

E.Esegui(
       delegate(int p)
       { Console.Write("...{0} ", p); }
     );

E.Esegui(  delegate(int p) { }  );
```

Espressioni Lambda (C# 3.0)

- Lambda Expressions
  - Funzioni anonime che contengono istruzioni C# e possono essere utilizzate per creare delegate anonimi.
  - Contengono l'operatore =>
    - La parte a sinistra dell'operatore => indica i parametri della funzione
    - La parte a destra contiene un'espressione o un insieme di istruzioni racchiuso fra parentesi graffe.

```
delegate int TipoFunzione(int parametro);
delegate int TipoFunzione2(int p1,int p2);
```

```
TipoFunzione f = x => x * x;
int a = f(5); // a = 25

TipoFunzione2 f2 = (x, y) => x + y;
int b = f2(3, 7); // b = 10
```

=

```
TipoFunzione f = delegate(int x)
                    { return x * x; };
int a = f(5); // a = 25

TipoFunzione2 f2 = delegate(int x, int y)
                    { return x + y; };
int b = f2(3, 7); // b = 10
```

```
E.Esegui(p => Console.Write("...{0} ", p));
```

=

```
E.Esegui( delegate(int p)
     { Console.Write("...{0} ", p); } );
```

Eventi

- Evento
  - Meccanismo con cui una classe può fornire notifiche quando qualcosa di interessante accade
  - Esempio: una classe che implementa un pulsante vorrà probabilmente segnalare quando questo viene premuto dall'utente
  - In C# gli eventi sono realizzati mediante delegate
- Esempio:

```
public delegate void ButtonEventHandler();

class TestButton
{ public event ButtonEventHandler OnClick;
  public void Click()
  {
    OnClick();
  }
}
...
TestButton mb = new TestButton();
mb.OnClick += new ButtonEventHandler(GestoreEvento);
```

```
void GestoreEvento()
{
 Console.WriteLine(
     "Pulsante premuto!“
 );
}
```

Namespace

- Utilizzati per contenere tipi (classi, strutture, interfacce,...) e altri namespace
- La Class Library è organizzata gerarchicamente mediante namespace
- La keyword using permette di evitare di specificare il namespace di ogni classe
- Esempi:

```
...
System.Console.WriteLine("Hello");
...
```

```
using System;
...
Console.WriteLine("Hello");
...
```

```
...
System.Windows.Forms.VisualStyles.BorderType s =
                System.Windows.Forms.VisualStyles.BorderType.Ellipse;
```

```
using System.Windows.Forms.VisualStyles;
...
BorderType s = BorderType.Ellipse;
```

Classi generiche (Generics)

- Parametri di tipo
  - È possibile progettare classi che rimandano la definizione di uno più tipi al momento in cui viene dichiarata o creata un'istanza della classe stessa.
  - Principali vantaggi:
    - Riutilizzo del codice
    - Aiutano a ridurre gli errori di programmazione attraverso un maggior controllo sui tipi
  - Possono essere creati anche metodi, interfacce e delegate generici
  - È possibile specificare dei vincoli sui parametri di tipo (mediante la parola chiave where)
- Esempio:

```
class Immagine<T> where T: struct
{
    public int Larghezza { get; private set; }
    public int Altezza { get; private set; }
    public Immagine(int w, int h)
    {
        Larghezza = w;
        Altezza = h;
        pixels = new T[w * h];
    }
    public T this[int i]
    {   get { return pixels[i]; }
        set { pixels[i] = value; }
    }
    private T[] pixels;
}
```

```
...
Immagine<byte> bImg = new Immagine<byte>(320, 200);
for (int i = 0; i < bImg.Altezza*Img.Larghezza; i++)
{
  bImg[i] = 255;
}

var cImg = new Immagine<Color>(640, 480);
for (int i = 0; i < cImg.Altezza * cImg.Larghezza; i++)
{
  cImg[i] = Color.White;
}
```

Generic collections

- Collections
  - La Class Library mette a disposizione classi che implementano le strutture dati più comuni
    - Liste, Code, Pile, Tabelle hash, …
  - Per la maggior parte di esse ne esistono due versioni:
    - Non-generic collections: utilizzano object come tipo di dato, in modo da poter essere utilizzate con qualsiasi tipo
      - Sono inefficienti con i value types, per le continue operazioni di boxing/unboxing
    - Generic collections: permettono di specificare il tipo dei dati al momento della dichiarazione
      - Sono più efficienti e in genere preferibili
- Esempio di utilizzo di due generic collections (lista e tabella hash):

```
List<Persona> Impiegati = new List<Persona>();
Impiegati.Add(new Persona("Guido", "Guidi"));
Impiegati.Add(new Persona("Carlo", "Bianchi"));
foreach (Persona P in Impiegati)
  Console.Write("{0}\n",P);
...
Dictionary<string, Studente> Idx = new Dictionary<string, Studente>();
Idx.Add("004536171", S1);
Idx.Add("004536172", S2);
Idx.Add("004536432", S3);
Studente s = Idx["004536172"];
```

Commenti XML

```
/// <summary>Documentazione generale della classe ... </summary>
class Persona : ICloneable, IFormattable
{
    /// <summary>Campi privati della classe</summary>
    private string Nome, Cognome;
    private DateTime DataDiNascita;
    /// <summary>
    /// Implementazione metodo Clone() dell'interfaccia ICloneable
    /// </summary>
    /// <seealso cref="System.ICloneable">
    /// <returns>Una nuova istanza con una copia dei dati</returns>
    public object Clone()
    ...
    /// <summary>
    /// Implementazione metodo ToString dell'interfaccia IFormattable
    /// </summary>
    /// <seealso cref="System.IFormattable">
    /// <param name="format">Formato: null - Nome,Cognome;
    /// "ALL" - Nome,Cognome,Data di nascita</param>
    /// <returns>I dati formattati come stringa</returns>
    public string ToString(string format, ...)
    ...
    /// <summary>Costruttore della classe</summary>
    public Persona(string nome, string cognome, ... )
    ...
```

Attributi

- Consentono di aggiungere metadati al codice
  - I metadati sono informazioni aggiuntive che possono essere associati a interi assembly, tipi, metodi, proprietà, parametri di metodi.
  - È possibile definire nuovi attributi oltre a quelli predefiniti (un attributo è semplicemente una classe derivata da System.Attribute)
  - Gli attributi possono essere esplorati a run-time mediante “reflection”

```
class UnaClasse
{
  [Obsolete("Sarà rimosso nella versione 2.0")]
  public int Somma(int a, int b)
  {
    return a + b;
  }
}
```

```
[AttributeUsage(AttributeTargets.Class)]
class Autore : Attribute
{
    public string Nome { get; private set; }
    public string Revisore { get; set; }
    public Autore(string nome)
    {  Nome = nome;
    }
}

[Autore("Mario Rossi")]
class Immagine<T> { ... }

[Autore("Mario Rossi", Revisore = "Carlo Bianchi")]
class Algoritmo<T> { ... }
```

Codice unsafe

- Normalmente il C# non prevede l'utilizzo di puntatori
  - Memoria gestita automaticamente, il garbage collector provvede a liberare quella non più in uso
  - Riduce la probabilità di introdurre errori e potenziali problemi di sicurezza
- Mediante la keyword unsafe è possibile definire un contesto (blocco di codice, metodo o tipo) in cui:
  - Poter dichiarare e utilizzare variabili di tipo puntatore (sintassi analoga al C)
  - Chiamare funzioni che richiedono l'utilizzo di puntatori
  - Eseguire operazioni aritmetiche sui puntatori
  - In pratica è "inline C"
- Nel Visual Studio è necessario impostare esplicitamente un'opzione nel compilatore per permettere codice unsafe

Approfondimenti

- Elementi non trattati in queste dispense ma utili per una conoscenza più generale del linguaggio C#:
  - Overloading degli operatori, operatori di conversione
  - Iterators
  - Ereditarietà e polimorfismo: keyword sealed, new virtual
  - Keyword readonly
  - using/dispose
  - params (funzioni con numero di parametri variabili)
  - Reflection
  - Nullable types
  - Interoperabilità (come chiamare codice unmanaged)
  - Multithreading e sincronizzazione
  - Tipi anonimi
  - LINQ, Expression Trees

C# – Risorse

- MSDN
  - Tutti gli articoli in “Visual C#”→ “Getting Started with Visual C#”
  - Tutti gli articoli in “Visual C#”→ “C# Programming Guide”

- Siti web
  - http://www.csharp-station.com
    - http://www.csharp-station.com/Tutorial.aspx
  - http://www.csharpcorner.com
  - http://www.codeproject.com
  - http://www.codeguru.com
  - http://www.codeplex.com