

C# FUNDAMENTALS

Come imparare a programmare e sopravvivere

#Start2DEV

Matteo Valoriani
mvaloriani@gmail.com

Obiettivi del corso

- Fornire le basi teoriche per sviluppare programmare con un linguaggio ad oggetti
- Fornire le basi pratiche per utilizzare il framework .Net
- Familiarizzare con i tool di sviluppo

WHO I AM...

PhD Student at Politecnico of Milano

CEO of Fifth Element



Microsoft Speaker

Consultant



mvaloriani@gmail.com
@MatteoValoriani



Follow me on
Twitter or the
Kitten gets it:
[@MatteoValoriani](#)

Contatti

MSP Exp - <https://www.facebook.com/MSPExp>

Evento - goo.gl/NTjyDE

Link utili

Libri e esempi:

- <http://csharpcourse.com/>
- [http://msdn.microsoft.com/it-it/library/aa287551\(v=VS.71\).aspx](http://msdn.microsoft.com/it-it/library/aa287551(v=VS.71).aspx)
- <http://www.microsoft.com/italy/beit/Default.aspx>
- <http://www.microsoftvirtualacademy.com/>

Software:

- <http://www.asict.polimi.it/software/microsoft/msdn-academic-alliance.html>
- <https://www.dreamspark.com/default.aspx>
- <http://weblogs.asp.net/sreejukg/archive/2011/01/06/documenting-c-library-using-ghostdoc-and-sandcastle.aspx>
- <http://grantpalin.com/2010/01/10/net-projects-generating-documentation-with-sandcastle/>

Programmazione a Oggetti

Prima della programmazione OO

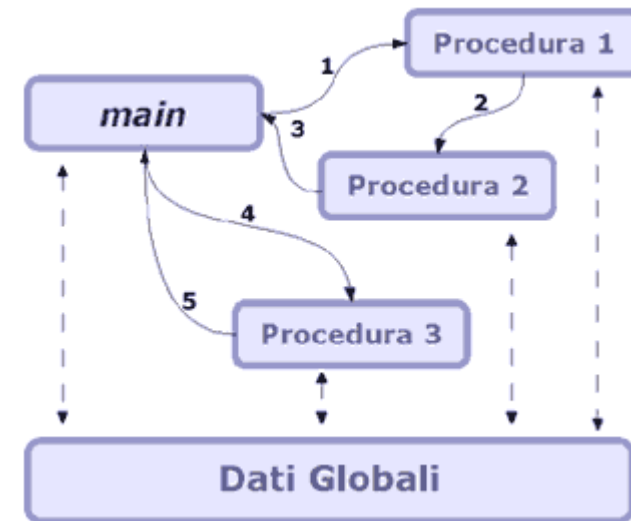
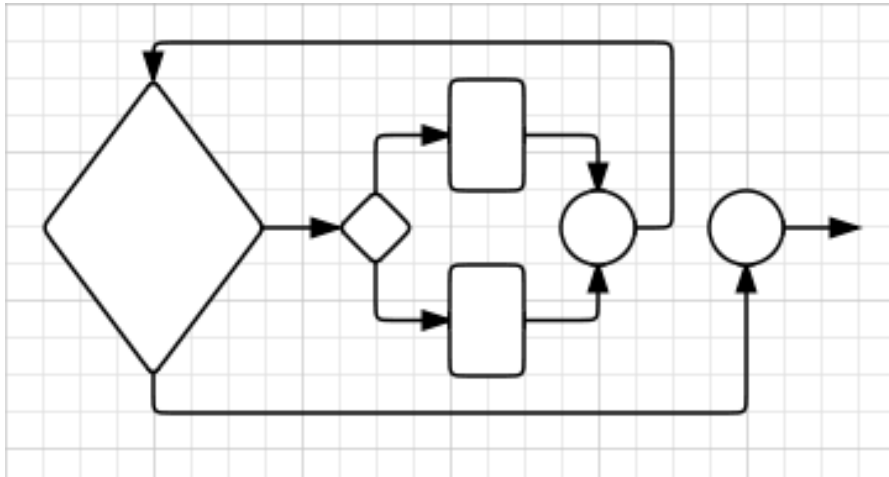
Anni '50 e inizio anni '60, programmazione a spaghetti:

Basata sul salto incondizionato GOTO

Anni '60, programmazione strutturata:

GOTO = ALTERNATIVA + RIPETIZIONE + SEQUENZA

Programma come workflow



Prima della programmazione OO

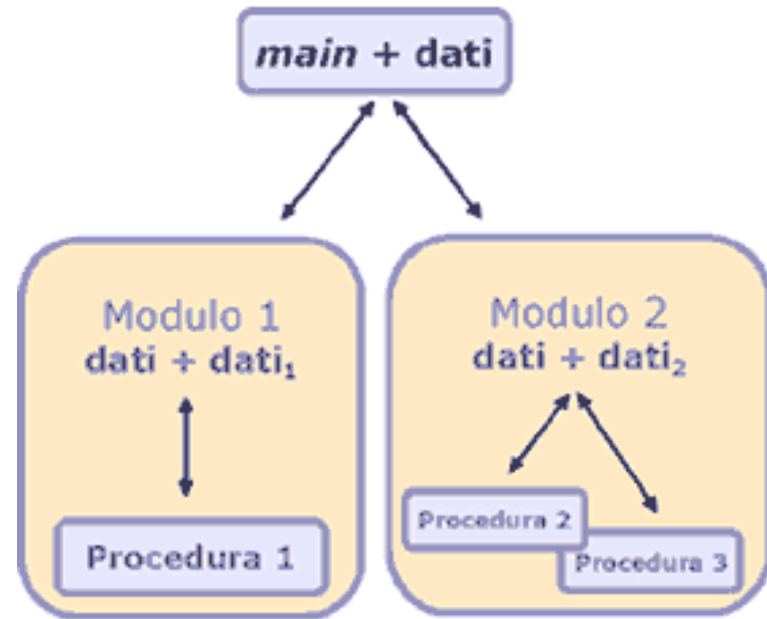
Anni '60, programmazione Modulare:

Programma composto da moduli che cooperano

Ogni modulo è indipendente

Moduli riusabili

Sviluppo separato



I primi linguaggi OO

Fine anni '60, ideata la programmazione a oggetti

Anni '70, nasce il primo linguaggio a oggetti SmallTalk

Anni '80, vengono sviluppati ADA e C++ e si diffonde la programmazione ad oggetto

Principi di Programmazione OO

Astrazione dell'idea di programma:

Prende spunto dal mondo «reale» per creare un mondo «virtuale»

Ogni programma è un insieme di «Oggetti» che interagiscono

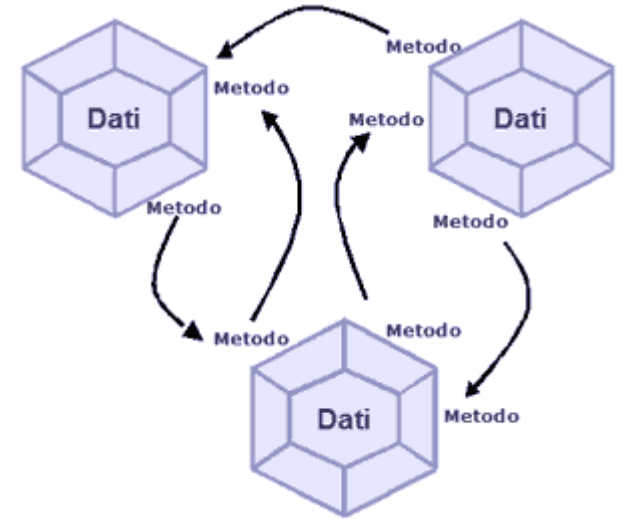
Le «Classi»:

Idea platonica della forma, la

Dichiarazione delle strutture dati interne

Operazioni, «metodi», che si possono eseguire sulla struttura dati

Le classi costituiscono dei modelli astratti, che a tempo di esecuzione sono istanziati per creare oggetti software. Questi ultimi sono dotati di proprietà e di metodi secondo quanto dichiarato dalle rispettive classi.



Oggetti

Stato di un oggetto

Lo stato rappresenta la condizione in cui si trova l'oggetto

Lo stato è definito dai valori delle variabili interne all'oggetto (proprietà)

Comportamento di un oggetto (behavior)

Determina come agisce e reagisce un oggetto

È definito dall'insieme di operazioni che l'oggetto può compiere (metodi)

Oggetto vs Classe

Compile Time

Studente.class

Studente

Run Time

Mario

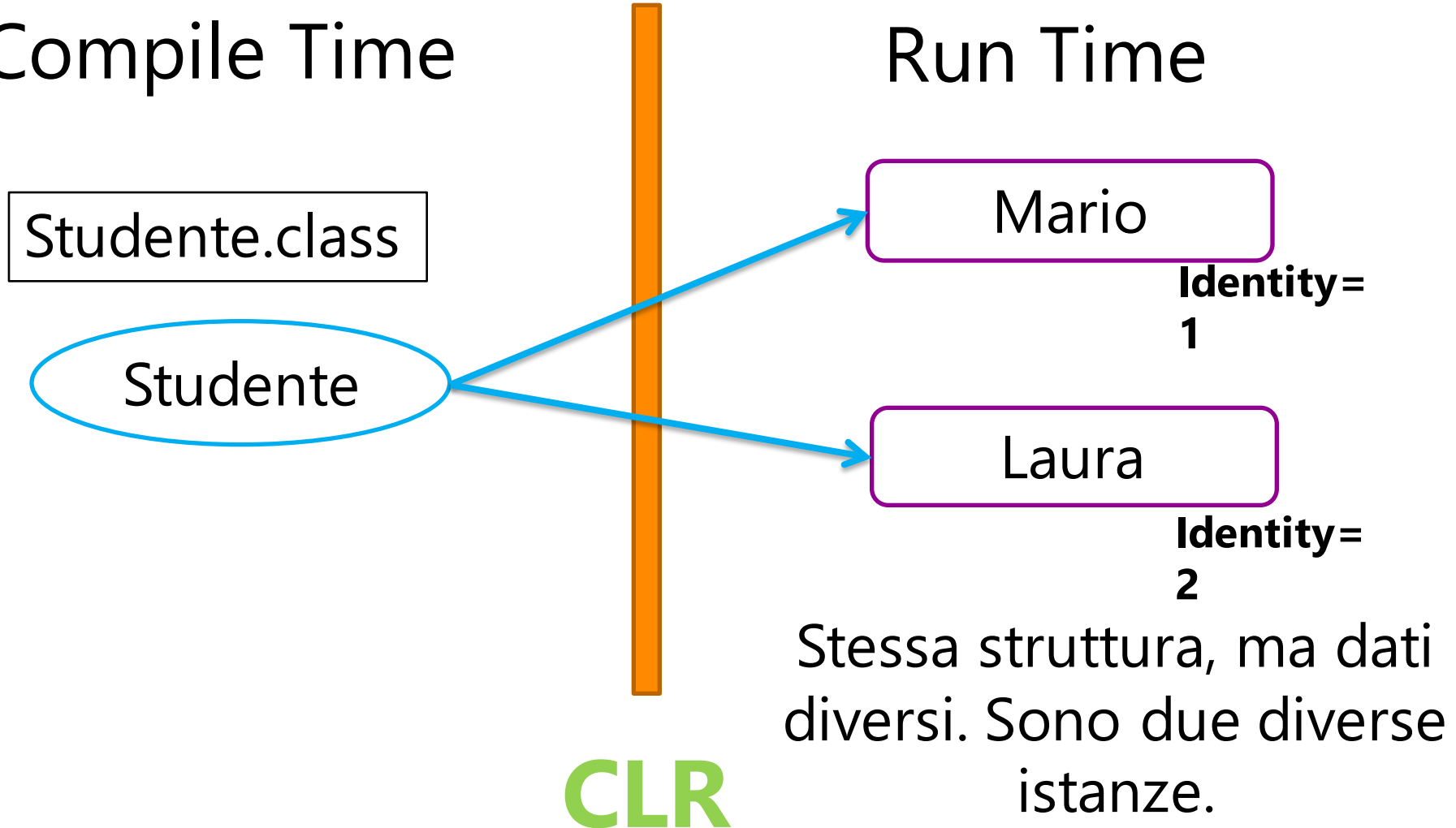
Identity=
1

Laura

Identity=
2

Stessa struttura, ma dati diversi. Sono due diverse istanze.

CLR



Istanze

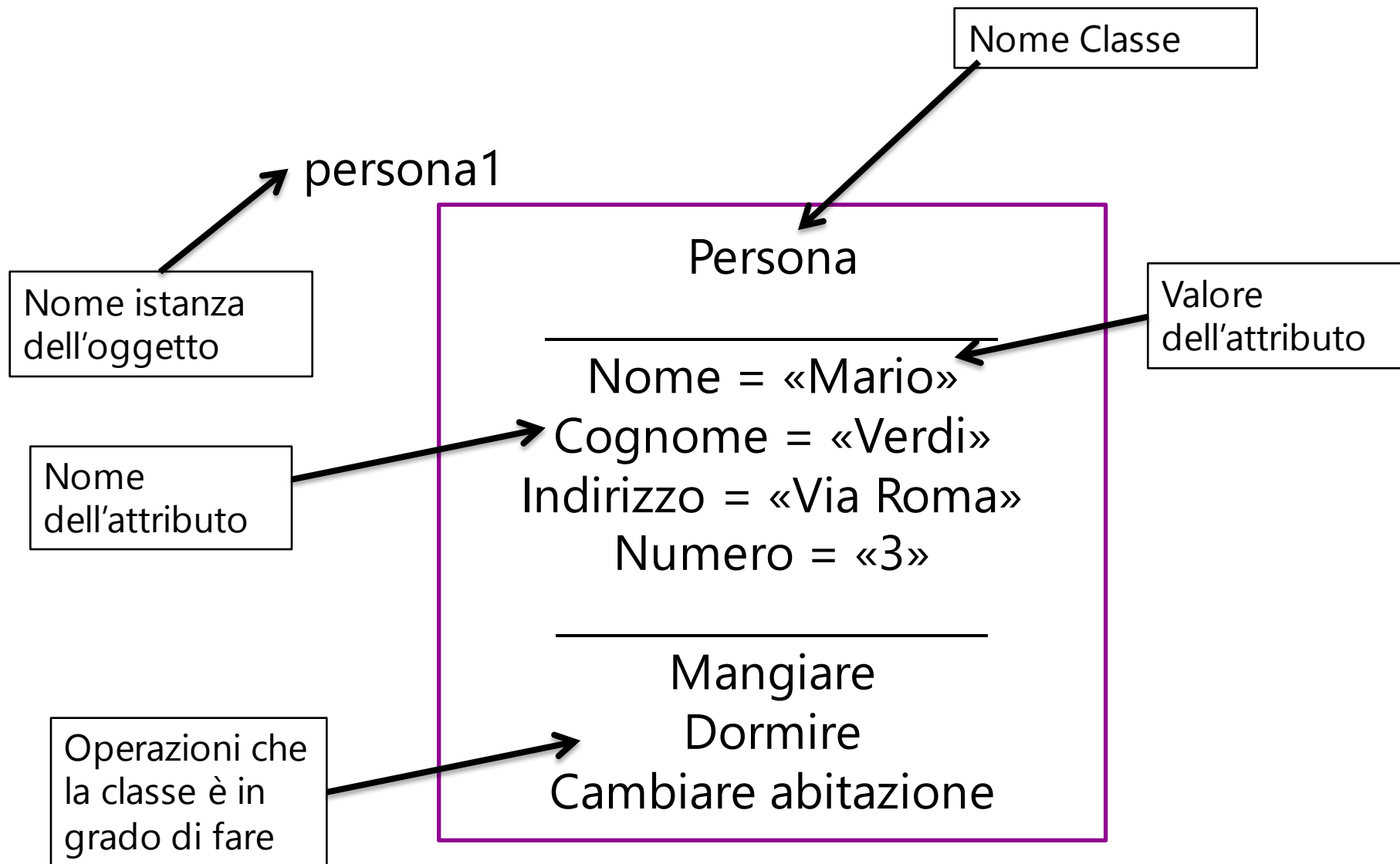


Diagramma delle classi

***caratteristiche
specifiche***

nome classe

attributo1
attributo2
attributo3
attributo4

***comportamenti
generali***

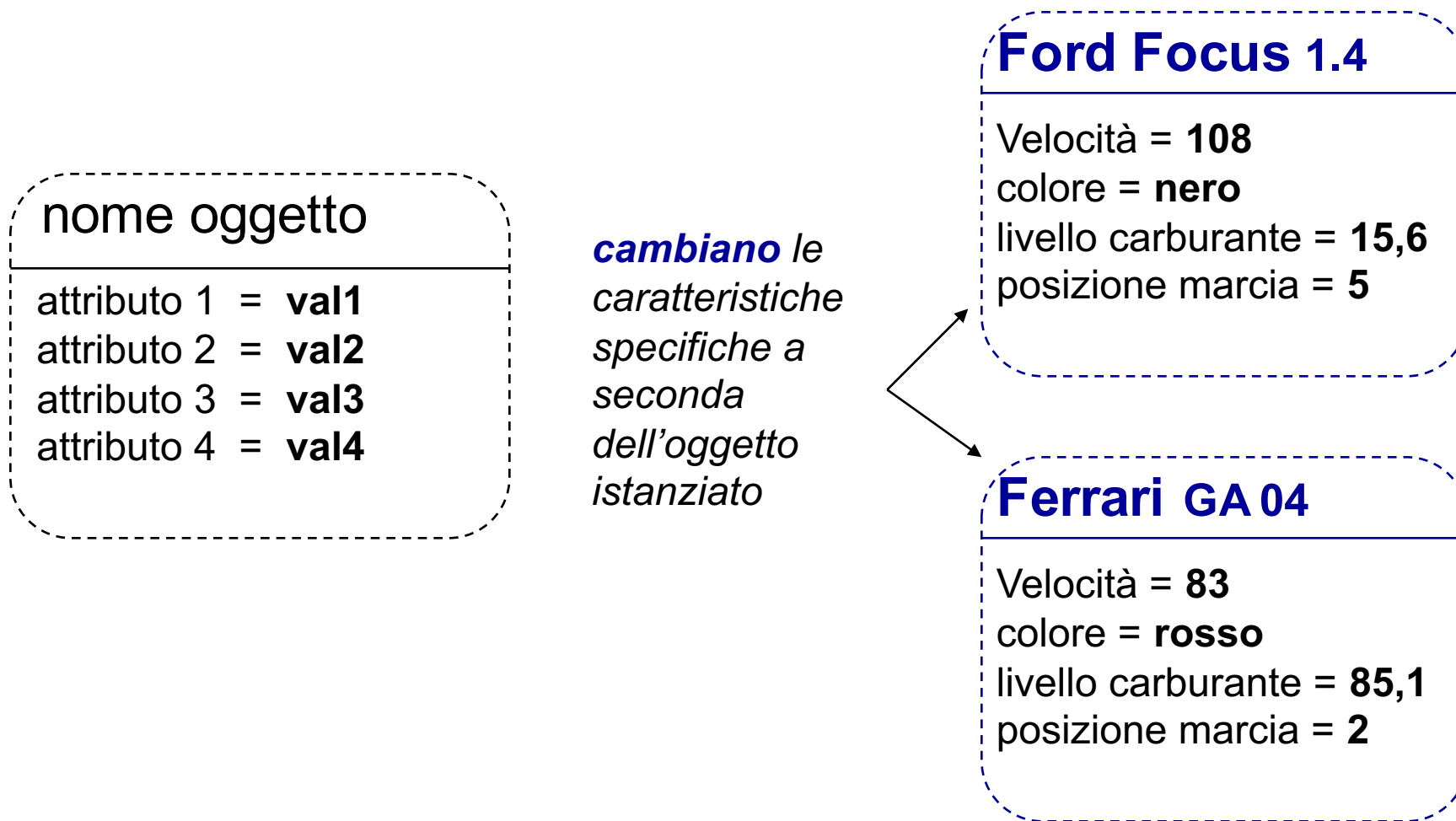
metodo1
metodo2
metodo3
metodo4
metodo5
metodo6

automobile

velocità
colore
livello carburante
posizione marcia

avviati
accelera
sterza
spegniti
cambia marcia
frena

Diagramma degli oggetti



Caratteristiche Linguaggio OO

Incapsulamento:

i dati che definiscono lo stato interno di un oggetto sono accessibili solo ai metodi dell'oggetto stesso.

Per alterare lo stato interno dell'oggetto, è necessario invocarne i metodi.

L'oggetto è come una black-box, cioè una «scatola nera» di cui attraverso l'interfaccia è noto cosa fa, ma non come lo fa.

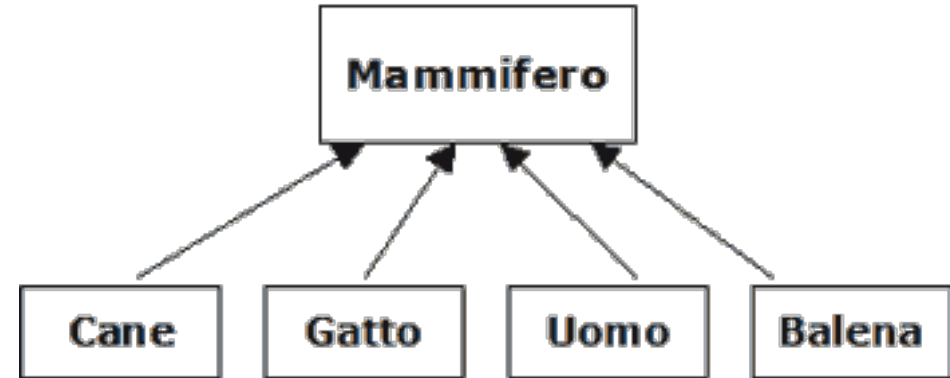
Information hiding, gestione della visibilità di quello che accade dentro la scatola

Caratteristiche Linguaggio OO (2)

Ereditarietà:

Permette di derivare nuove classi a partire da quelle già definite.

Una classe derivata, sottoclasse, mantiene i metodi e gli attributi delle classi da cui deriva (classi base, o superclassi)



La sottoclasse può definire i propri metodi o attributi e può ridefinire i metodi ereditati (overriding)

Quando una classe eredita da una sola superclasse si parla di eredità singola; viceversa, si parla di eredità multipla.

Meccanismo per ottenere l'estensibilità e il riuso del codice

Caratteristiche Linguaggio OO (3)

Polimorfismo:

Le istanze di una sottoclasse possono essere utilizzate al posto di istanze della superclasse

I metodi che vengono ridefiniti in una sottoclasse sono detti polimorfi, in quanto lo stesso metodo si comporta diversamente a seconda del tipo di oggetto su cui è invocato.

Altre keyword

Interfaccia:

Tipi e operazioni definiti in un componente che sono visibili fuori del componente stesso.

Specifica:

Funzionamento del componente, espresso mediante proprietà osservabili attraverso l'interfaccia.

Implementazione:

Strutture dati e funzioni definiti dentro al componente, non necessariamente visibili da fuori.

Pensare ad oggetti

In un programma di tipo procedurale, si è soliti iniziare a ragionare in maniera top-down, partendo cioè dal main e creando mano a mano tutte le procedure necessarie.

Pensare ad oggetti vuole dire identificare gli oggetti che entrano in gioco nel programma che vogliamo sviluppare e saperne gestire l'interazione degli uni con gli altri.

Nella programmazione ad oggetti serve definire prima le classi e poi associare ad esse le proprietà ed i metodi opportuni.

Come si scelgono gli oggetti?

Cosa si vuole che sia in grado di fare?

Un oggetto che abbia uno o due soli metodi deve fare riflettere

Oggetti senza o con troppi metodi sono, in genere, da evitare

Quali proprietà sono necessarie affinché l'oggetto sia in grado di eseguire le proprie azioni?

Attributi, proprietà che descrivono le caratteristiche peculiari di un oggetto (peso, altezza, ...)

Componenti, proprietà che sono atte a svolgere delle azioni (testa, mani, piedi, ...)

Peer objects, proprietà che a loro volta sono identificate e definite in altri oggetti (auto di una persona,...)

Accoppiamento & Coesione

Accoppiamento: quanto sono legate (dipendenze reciproche) due unità separate di un programma.

Coesione: quantità e eterogeneità dei task di cui una singola unità (una classe o un metodo) è responsabile. Se una classe ha una responsabilità ristretta ad un solo compito il valore della coesione è elevato.

Obiettivo: alta coesione e basso accoppiamento

C#

Storia (solo un po')

In principio fu la luce

Gennaio 1999, [Anders Hejlsberg](#) crea un team per realizzare un nuovo linguaggio chiamato Cool, C-like Object Oriented Language

Gennaio 2002, il framework .Net è disponibile per il download e il linguaggio è rinominato C#

Settembre 2005, esce la v.2, C# diventa una valida alternativa a Java

Agosto 2007, esce la v.3 e successivamente la 3.5 (attualmente la più usata) che introducono elementi funzionali nel linguaggio

Aprile 2010, esce la v.4

Caratteristiche di C#

Multi-paradigm programming language

Language interoperability(C/C++, VB,...)

Interoperabilità via standard web(SOAP, XML,...)

Linguaggio Object Oriented (Tutto è DAVVERO un oggetto)

Framework .Net

Ambiente di programmazione orientato agli oggetti coerente

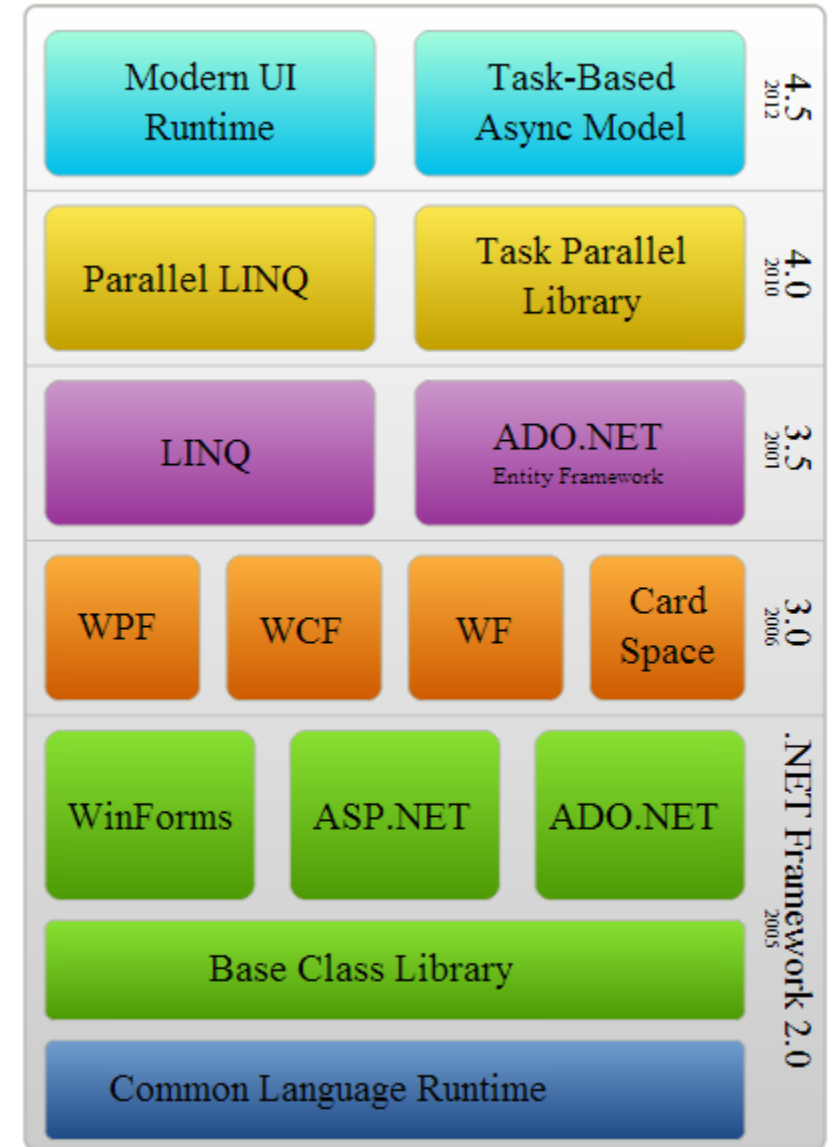
Minimizzi la distribuzione del software e i conflitti di versioni.

Esecuzione sicura anche dei codici creati da produttori sconosciuti o semi-trusted.

Eliminare i problemi di prestazioni degli ambienti basati su script o interpretati.

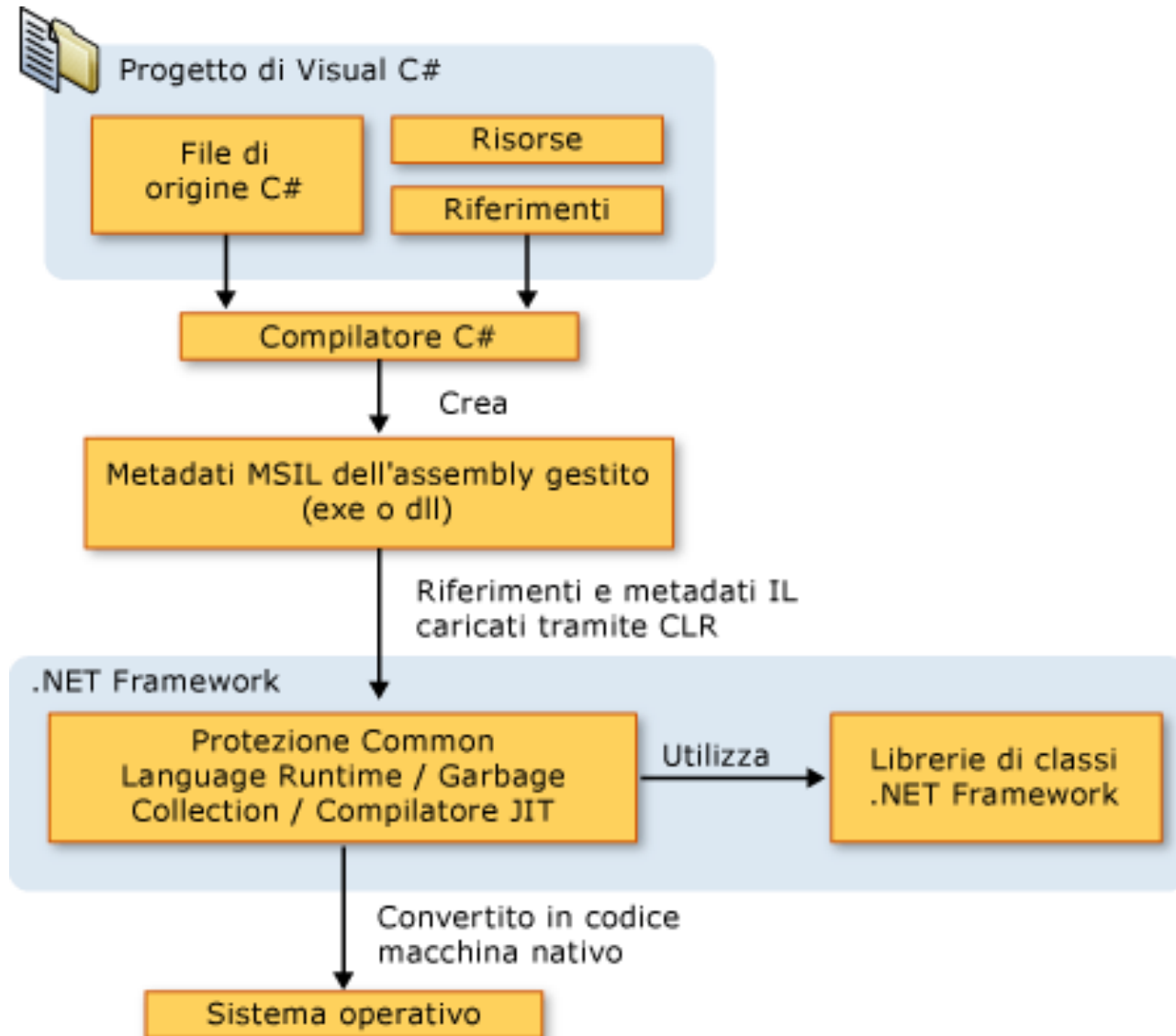
Rendere coerente l'esperienza dello sviluppatore attraverso tipi molto vari di applicazioni (web, windows, mobile)

Generare tutte le comunicazioni in base agli standard industriali per assicurare che il codice basato su .NET Framework possa integrarsi con qualsiasi altro codice.



The .NET Framework Stack

Architettura



Ora iniziamo veramente

Visual studio 2013

Integrated Development Environment (IDE)

Supporto a molti linguaggi: C, C++, C#, F#, Visual Basic .Net e ASP .Net

Integra la tecnologia IntelliSense la quale permette di correggere errori prima ancora di compilare

Visual Studio 2013 (2)

IntelliSense: completamento automatico del codice che si attiva non appena si inizia a digitare qualcosa nell'editor.

Refactoring: con questo termine si intende una serie di funzionalità che permettono di modificare velocemente e sistematicamente il codice scritto.

Code Snippets e Surrounding: gli "snippet" sono porzioni di codice di uso comune, come il codice dei costrutti for e while. Clic col tasto destro del mouse nella finestra del codice e selezionando il comando Insert Snippet..., oppure digitando il nome dello snippet e premendo due volte il tasto TAB

Console Application

No componenti visuali (buttons, text boxes, etc.)

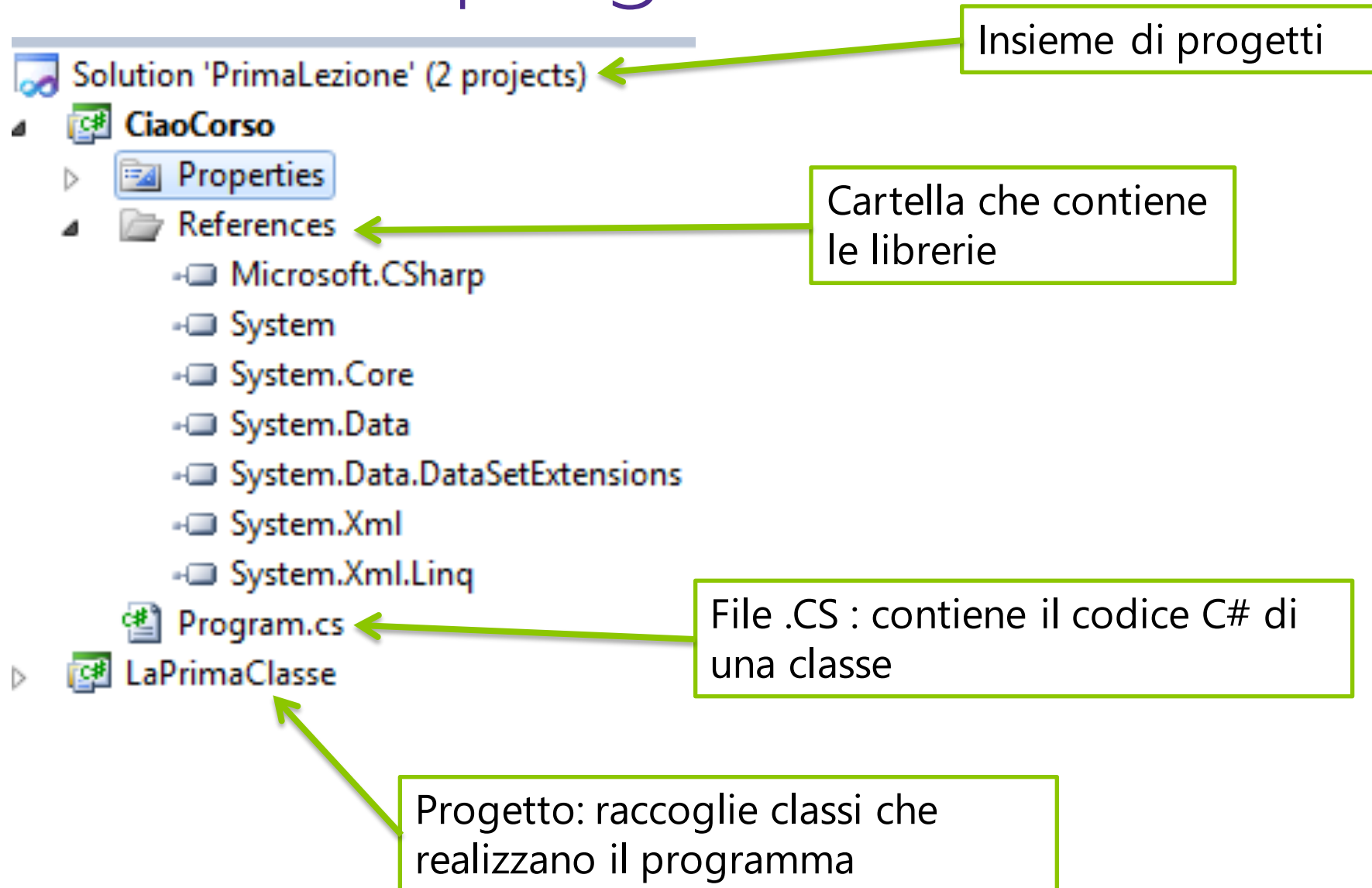
Solo output di testo

Due tipi:

MS-DOS prompt - Windows 95/98/ME

Command prompt - Windows 2000/NT/XP

Struttura di un progetto



Ciao Corso

```
using System;  
  
namespace CiaoCorso  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // Scrive sulla console  
            System.Console.WriteLine("Ciao Corso");  
            // Aspettata che la persona digiti un carattere  
            System.Console.ReadKey();  
        }  
    }  
}
```

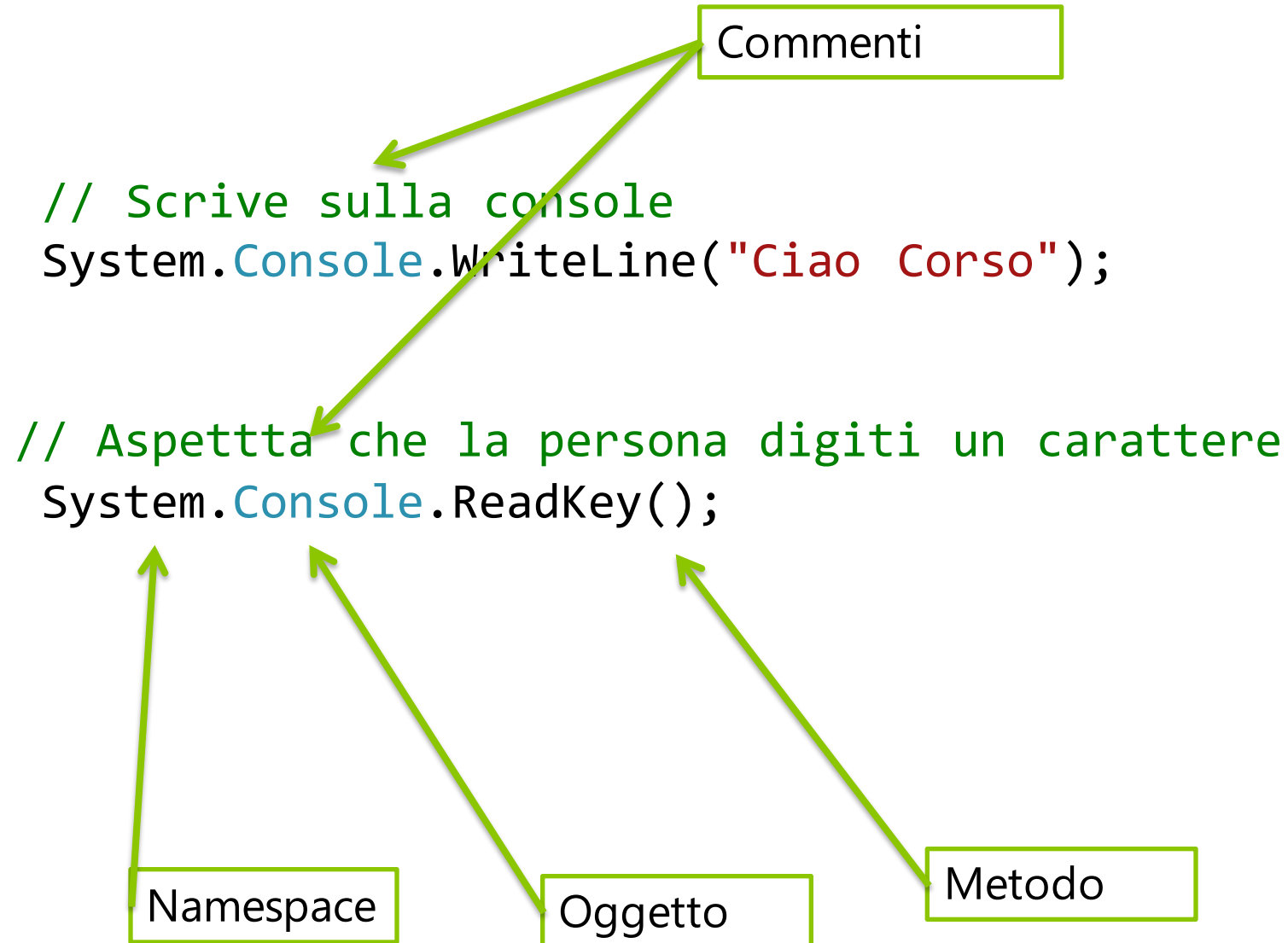
Indica dove prendere le librerie usate

Metodo per raggruppare le funzionalità

Nome della classe

Metodo speciale: punto di ingresso dell'applicazione. Ogni applicazione ne deve avere uno.

Ciao Corso (2)



Strutture base: Cicli

For:

```
for (int i = 0; i < length; i++)  
    { //fai qualcosa }
```


Numero fisso di volte



While:


```
while (condizione==true)  
    { //fai qualcosa }
```

Finche la condizione è vera il
ciclo continua



Do:

```
do{ //fai qualcosa  
    } while (condizione== true);
```



Strutture base: Cicli(2)

For:

```
foreach (Persona item in listaPersone)
{
    item.ChiSei();
}
```

Alternative

If

```
if (condizione == true)
    { //fai qualcosa }
else
    { //fai qualcosa }
```

Switch

```
switch (switch_on) {
    case 1: //fai qualcosa
        break;
    case 2: //fai qualcosa
        break;
    default: //fai qualcosa
}
```

Creare una classe

```
class NomeClasse
{
    //Proprietà
    visibilità tipo nomeProprietà;

    //Costruttore
    public NomeClasse() { }

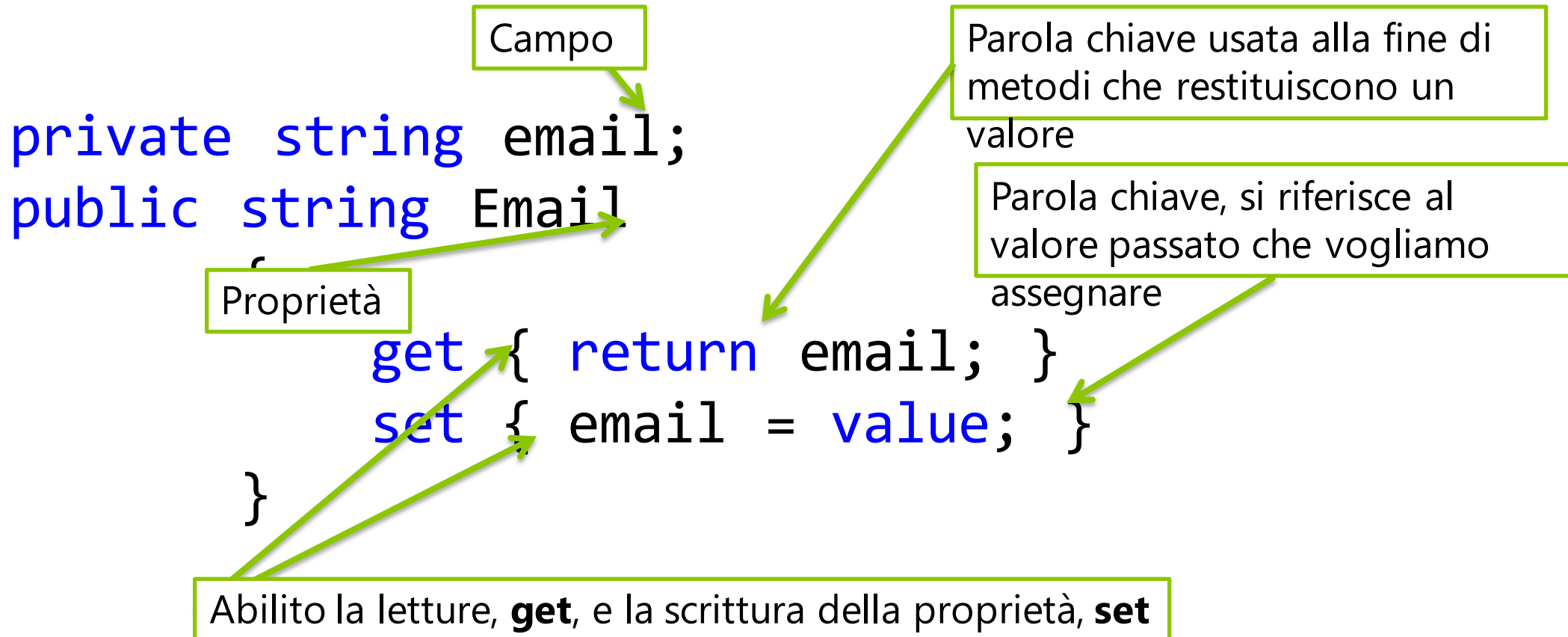
    //Metodi
    visibilità tipoRitornato nomeMetodo1() { }
}
```

Modificatori di accesso

public	Nessuna restrizione di accesso.
protected	L'accesso è limitato alla classe di appartenenza o ai tipi derivati dalla classe di appartenenza.
internal	L'accesso è limitato all'assembly corrente.
protected internal	L'accesso è limitato all'assembly corrente o ai tipi derivati dalla classe di appartenenza.
private	L'accesso è limitato al tipo di appartenenza.

Campi e Proprietà

Le proprietà consentono a una classe di esporre un modo pubblico per ottenere e impostare valori, nascondendo tuttavia il codice di implementazione o di verifica.



Metodi

«Azioni» che permettono di interagire con gli oggetti

```
public string ChiSei() {  
    string result = "";  
    result = "Io sono " + nome + " " + cognome + "/n";  
    result = result + "La mia email è " + email;  
    return result;  
}
```

Tipo restituito

Creo una stringa vuota

Il simbolo + viene usato per
«concatenare», cioè unire, le stringhe

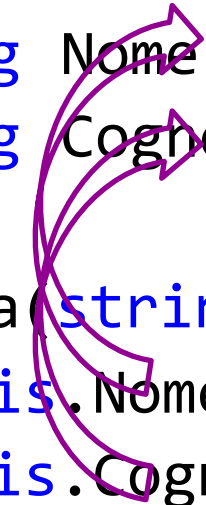
Alla stringa iniziale concatenano altre
parti

Valore restituito dalla funzione

Costruttori

Servono per creare l'istanza della classe, cioè l'oggetto usato durante l'esecuzione del programma

```
private string Nome;  
private string Cognome;  
  
public Persona(string Nome, string Cognome) {  
    this.Nome = Nome;  
    this.Cognome = Cognome;  
}
```

A diagram consisting of three purple curved arrows. The first arrow starts from the parameter 'Nome' in the constructor signature and points to the class attribute 'Nome'. The second arrow starts from the parameter 'Cognome' in the constructor signature and points to the class attribute 'Cognome'. The third arrow starts from the 'this' keyword in the first assignment statement and points to the class attribute 'Nome'.

La keyword **this** permette di risolvere i problemi di nomi

Overload

Metodo con stesso nome, ma con parametri diversi o con valore di ritorno diverso

```
public Persona(string Nome, string Cognome ) {  
    this.Nome = Nome;  
    this.Cognome = Cognome;  
}
```

```
public Persona(string nome, string cognome, string email)  
{  
    this.nome = nome;  
    this.cognome = cognome;  
    this.email = email;  
}
```

Utilizzo dei metodi

```
static void Main(string[] args)
{
    Persona mario = new Persona("Mario", "Verdi");

    mario.Email = "mario@polimi.it";

    string rit = mario.ChiSei();
    Console.Write(rit);
    Console.ReadKey();
}
```

La keyword **new**, serve per creare nuovi oggetti. Si usa prima del costruttore

Le proprietà si utilizzano semplicemente con = senza le ()

Classica chiamata di un metodo con valore di ritorno

Ereditarietà

Permette di estendere classi già esistenti con nuove funzionalità, senza dover riscrivere tutto il codice

```
class Studente : Persona  
{ ... }
```

Nuova classe

Classe da cui eredita

- La classe Studente(S) è effettivamente sia Studente che Persona(P).
- È possibile utilizzare un'operazione di «**cast**» per convertire S in un oggetto P. L'operazione di cast non implica la modifica dell'oggetto S, ma limita semplicemente la visualizzazione dell'oggetto S ai dati e ai comportamenti dell'oggetto P
- Dopo aver eseguito il cast di un oggetto S in un oggetto P, è possibile eseguire il cast dell'oggetto P per convertirlo di nuovo in S.
- Non è possibile fare il cast da P a S se l'oggetto non è un'istanza di S.

Richiamo costruttore base

```
private int matricola;
```

```
public Studente(string nome, string cognome,  
                int matricola) : base(nome, cognome) {  
    this.matricola = matricola;  
}
```

- La keyword «**base**» che richiama i metodi della classe base, in questo caso direttamente il costruttore, e gli passa i parametri
- Dopo l'esecuzione del costruttore della classe base, vengono eseguite le operazioni rimanenti.

Richiamo costruttore base(2)

```
private int matricola;  
public Studente(string nome, string cognome,  
                int matricola) : base(nome, cognome) {  
    this.matricola = matricola;  
}
```

- La keyword «**base**» che richiama i metodi della classe base, in questo caso direttamente il costruttore, e gli passa i parametri
- Dopo l'esecuzione del costruttore della classe base, vengono eseguite le operazioni rimanenti.

```
public Studente(string nome, string cognome) :  
    this(nome, cognome, 00000) { }
```

- In questo caso il costruttore richiama il costruttore definito prima settando un valore di default per la matricola

Override

- Sostituisco un metodo della classe base

```
public virtual string ChiSei()
```

Nella classe base aggiungo la parola
«**virtual**»

```
public override string ChiSei() {  
    string result = base.ChiSei();  
    result = result + "\n" + "La mia matricola è " +  
matricola;  
    return result;  
}
```

Nella sottoclasse aggiungo la parola «**override**»
prima del metodo

Override(2)

Metodo alternativo:

```
public new string ChiSei() {  
    string result = base.ChiSei();  
    result = result + "\n" + "La mia matricola è " +  
matricola;  
    return result;  
}
```

Nella sottoclasse aggiungo la parola «**new**» prima del metodo

- In realtà new nasconde la definizione della classe base con lo stesso nome
- Il comportamento risultante è diverso rispetto a override

new Vs override

```
Studente mario = new Studente("Mario", "Verdi");  
mario.Email = "mario@polimi.it";  
string rit = mario.ChiSei();  
Console.Write(rit);
```

```
Console.Write("\n\n");
```

```
rit = (mario as Persona).ChiSei();  
Console.Write(rit);  
Console.ReadKey();
```

Casting, uso mario come un oggetto
Persona



new Vs override

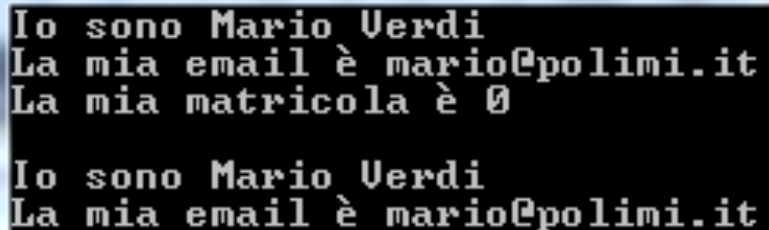
Se uso new

```
DerivedClass B = new DerivedClass();  
B.DoWork();
```

// Calls the new method.

```
BaseClass A = (BaseClass)B;  
A.DoWork();
```

// Also calls the old method.



```
Io sono Mario Verdi  
La mia email è mario@polimi.it  
La mia matricola è 0  
  
Io sono Mario Verdi  
La mia email è mario@polimi.it
```

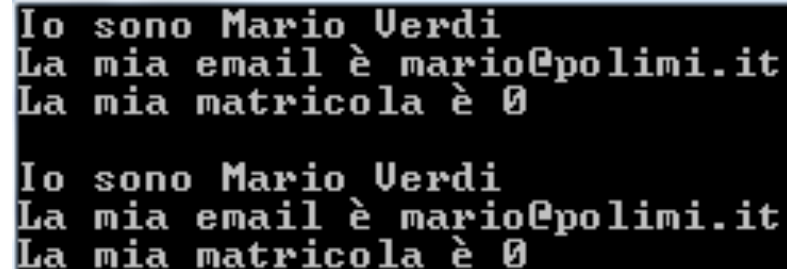
Se uso virtual+override

```
DerivedClass B = new DerivedClass();  
B.DoWork();
```

// Calls the new method.

```
BaseClass A = (BaseClass)B;  
A.DoWork();
```

// Also calls the new method.



```
Io sono Mario Verdi  
La mia email è mario@polimi.it  
La mia matricola è 0  
  
Io sono Mario Verdi  
La mia email è mario@polimi.it  
La mia matricola è 0
```

Classi e metodi «sealed»

```
public sealed class D { // Class members here. }
```

- Una classe **sealed** non può essere utilizzata come classe base.

```
public class D : C {  
    public sealed override void DoWork() { }  
}
```

La parola chiave **sealed** prima della parola chiave **override** nega l'aspetto virtuale del membro per qualsiasi ulteriore classe derivata. Non si può utilizzare override nelle sottoclassi.