

# **OOP: oggetti I**

**Dalla programmazione procedurale a quella  
object oriented**

Anno 2018/2019

## Indice generale

<b>1</b>	<b>Introduzione: (<i>programmazione procedurale</i>).....</b>	<b>3</b>
1.1	Programmazione “procedurale” .....	3
1.2	I “record” .....	4
1.3	I “moduli” .....	4
1.4	Conclusioni.....	5
<b>2</b>	<b>Implementare una “pila” (col modello procedurale).....</b>	<b>6</b>
2.1	Implementare la pila mediante un modulo.....	6
2.2	Vantaggi e limiti dell’uso di un modulo.....	7
<b>3</b>	<b>Superare il limite dei moduli: moduli + record.....</b>	<b>8</b>
3.1	Vantaggi e limiti dell’uso di modulo+record.....	9
<b>4</b>	<b>Programmazione OO col modello procedurale.....</b>	<b>10</b>
4.1	“Fondere” modulo e record.....	10
4.2	Conclusioni.....	11
<b>5</b>	<b>Modello di programmazione object oriented.....</b>	<b>12</b>
5.1	Uso degli oggetti.....	13
5.2	Accesso all’oggetto nei metodi di istanza: parola chiave “this” .....	13
5.3	“Status” speciale dei costruttori.....	14
<b>6</b>	<b>Oggetti “svelati” (in linguaggio assembly).....</b>	<b>15</b>
6.1	Classe Punto.....	15
6.2	Analisi del codice corrispondente in linguaggio <i>assembly</i> .....	16
6.2.1	Codice assembly corrispondente alla classe Punto.....	17
6.3	Conclusioni.....	18

# 1 Introduzione: (programmazione procedurale)

Questo tutorial fornisce un'introduzione alla programmazione *object oriented*, con l'obiettivo di svelare la natura dei suoi costituenti fondamentali: gli *oggetti*. Sarà un percorso:

- Partirò da un riepilogo sulla *programmazione procedurale*, prendendo in considerazione anche *record* e *moduli*.
- Mediante un semplice esempio, mostrerò i limiti di questo modello di programmazione.
- Mostrerò come superarli, utilizzando un modello di programmazione orientato agli oggetti, ma sempre restando all'interno della cornice procedurale del linguaggio.
- Infine, mostrerò l'equivalenza tra il modello precedente e il modello di programmazione *object oriented* adottato dal C#.

## 1.1 Programmazione “procedurale”

Da wikipedia:

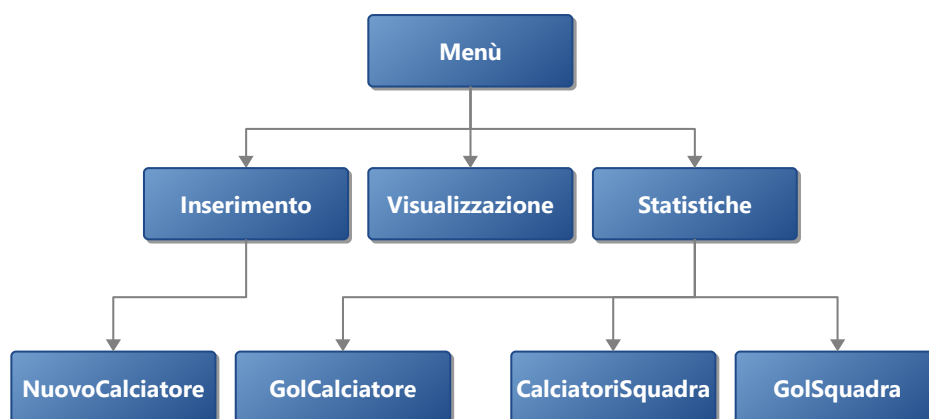
*La programmazione procedurale è un paradigma di programmazione che consiste nel creare dei blocchi di codice, identificati da un nome... Questi sono detti sottoprogrammi, procedure o funzioni, a seconda del linguaggio...*

In sostanza, il programma è strutturato in procedure (in C#, metodi) ognuna delle quali risolve un sotto problema.

Nonostante la sua semplicità, questo modello consente di produrre programmi con strutture complesse, suddividendoli in:

- Metodi che implementano requisiti altamente specifici del problema.
- Metodi che implementano requisiti di livello più grande.
- Metodi che implementano procedimenti generalizzabili, riutilizzabili in più applicazioni.

Ad esempio, nel caso di un'applicazione che debba gestire il campionato di calcio, si potrebbe immaginare la seguente struttura (ogni blocco è un metodo):



## 1.2 I “record”

I *record* esistono in tutti i modelli di programmazione, e sono fondamentali, poiché consentono di andare oltre la rappresentazione di semplici dati (numeri, stringhe, caratteri, etc), e di ragionare in termini di *concetti*, o *entità*.

Nell'esempio precedente, invece di realizzare strutture dati contenenti i nomi dei calciatori, i gol segnati, i nomi delle squadre in cui militano, etc, diventa possibile rappresentare il concetto di *calciatore*:

```
class Calciatore
{
    public string Nome;
    public string Cognome;
    public string Squadra;
    public int Gol;
}
```

Ciò semplifica la scrittura del programma, nella gestione dei dati e nell'implementazione dei metodi, soprattutto quelli che non hanno bisogno di elaborare i singoli dati di un calciatore (l'inserimento e all'eliminazione di un calciatore, ad esempio).

## 1.3 I “moduli”

Nella sua forma di base, la sola unità di codice del modello procedurale è il metodo. Ma quando la complessità del codice comincia a crescere, questo modello presenta delle problematiche:

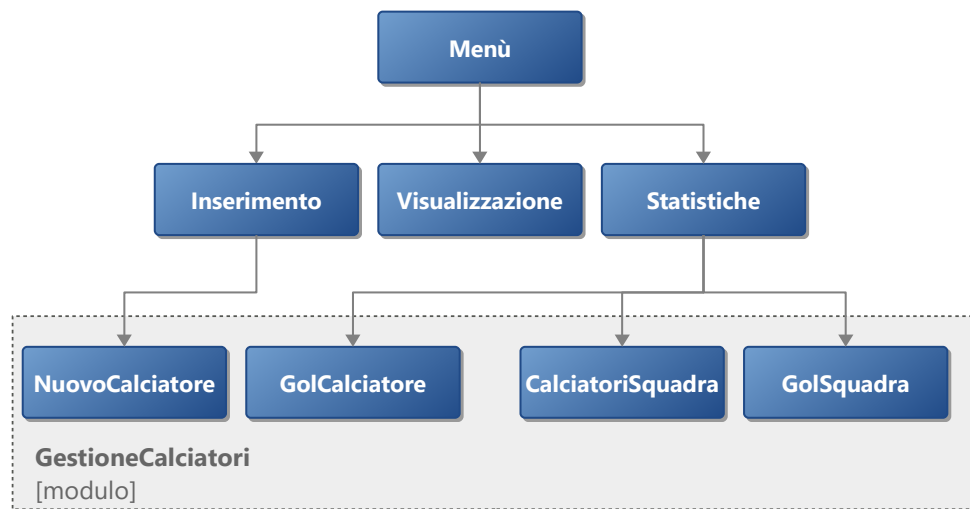
- La semplice organizzazione: avere tutti i metodi nello stesso contenitore pone problemi di gestibilità del codice.
- La difficoltà nella gestione di funzioni di livello superiore dell'applicazione, che richiedono la collaborazione di più metodi.
- La difficoltà nella gestione di funzioni di livello superiore generali, potenzialmente riutilizzabili in altre applicazioni.
- L'impossibilità di “incapsulare” queste funzioni, cosa che consentirebbe la loro modifica senza rischiare di produrre effetti indesiderati sul resto del codice.

I *moduli*, unità di codice di livello superiore ai metodi, permettono di affrontare con successo queste problematiche.

Ad esempio, il modulo `Math` semplifica l'uso delle funzioni matematiche di base, collocandole dentro un unico contenitore. La maggior parte dei metodi che definisce (ma non tutti!) sono indipendenti gli uni dagli altri.

Il modulo `Console` implementa la gestione dell'input/output di un'applicazione console. Definisce dati e metodi altamente integrati tra loro, poiché collaborano al perseguimento dello stesso obiettivo.

Infine, considerando nuovamente il programma sul campionato di calcio, si potrebbe pensare di incapsulare l'elaborazione dei calciatori in un modulo, separandola dalla gestione dell'interfaccia utente:



## 1.4 Conclusioni

Dopo aver riepilogato le caratteristiche principali del modello di programmazione procedurale, nei prossimi capitoli lo utilizzerò con un problema di programmazione che metterà in evidenza i suoi limiti.

Per coerenza, negli esempi di codice mi limiterò a usare il C# a livello procedurale, ma non potrò farlo completamente, poiché userò i vettori, che in C# sono oggetti.

## 2 Implementare una “pila” (col modello procedurale)

Supponiamo, nell’ambito di un’applicazione, di aver bisogno di una struttura LIFO di interi. Questa definisce come minimo due operazioni: *push* → inserisce un elemento; *pop* → estrae l’ultimo elemento inserito.

L’approccio corretto suggerisce di separare questa funzionalità dal resto dell’applicazione, incapsulandola in un modulo.

### 2.1 Implementare la pila mediante un modulo

```
static class Stack
{
    private static int[] items;
    private static int head;

    public static void Create(int capacity)
    {
        items = new int[capacity];
        head = -1;
    }

    public static void Push(int item) //-> IndexOutOfRangeException se la pila è piena
    {
        head++;
        items[head] = item;
    }

    public static int Pop() //-> IndexOutOfRangeException se la pila è vuota
    {
        return items[head--];
    }

    public static bool IsEmpty()
    {
        return head == -1;
    }

    public static bool IsFull()
    {
        return head == items.Length-1;
    }
}
```

Nota bene: ho implementato una pila con capacità fissa. I metodi `Push()` e `Pop()` sollevano un’eccezione se si tenta di inserire un elemento e la pila è piena, oppure si tenta di estrarlo e la pila è vuota.

Segue un frammento di codice che usa il modulo:

```

static void Main(string[] args)
{
    Stack.Create(5); // -> crea pila con una capacità di 5 elementi

    Stack.Push(10);
    Stack.Push(20);
    Stack.Push(30);

    while (Stack.IsEmpty() == false)
    {
        int item = Stack.Pop();
        Console.WriteLine(item);
    }
    //-> 30, 20, 10
}

```

## 2.2 Vantaggi e limiti dell'uso di un modulo

Realizzare la pila mediante un modulo fornisce i vantaggi dell'incapsulamento:

- Usare una funzionalità senza dipendere dal modo in cui è implementata. (E dunque poter cambiare l'implementazione senza modificare il comportamento del programma.).
- Semplificare l'uso della funzionalità.
- Riutilizzare la funzionalità in altri contesti / applicazioni.

Ma comporta anche un grosso limite: i campi di un modulo sono univoci all'interno del programma, *ciò significa che non è possibile usare due pile contemporaneamente.*

Si tratta di un limite inaccettabile per una pila, come per qualsiasi altra struttura dati. (Si immagini, nei propri programmi, di poter disporre soltanto di *una* pila, *una* coda, *una* lista, *un* dizionario, etc.)

### 3 Superare il limite dei moduli: moduli + record

Avere più pile significa poter disporre di più coppie dei campi `items` e `head`. Per ottenere questo occorre spostare i campi dal modulo, separando i dati dai metodi che li elaborano:

```
class StackData
{
    public int[] items;
    public int head;
}
```

Ogni metodo del modulo dichiarerà come primo parametro uno `StackData` ed eseguirà su di esso le operazioni necessarie:

```
static class Stack
{
    public static void Create(StackData stack, int capacity)
    {
        stack.items = new int[capacity];
        stack.head = -1;
    }
    public static void Push(StackData stack, int item)
    {
        stack.head++;
        stack.items[stack.head] = item;
    }

    public static int Pop(StackData stack)
    {
        return stack.items[stack.head--];
    }

    public static bool IsEmpty(StackData stack)
    {
        return stack.head == -1;
    }

    public static bool IsFull(StackData stack)
    {
        return stack.head == stack.items.Length - 1;
    }
}
```

Segue un frammento di codice che dimostra l'uso contemporaneo di due pile:

```
static void Main(string[] args)
{
    StackData stack1 = new StackData();
    StackData stack2 = new StackData();

    Stack.Create(stack1, 5); // -> crea pila con una capacità di 5 elementi
}
```



```

Stack.Push(stack1, 10);
Stack.Push(stack1, 20);
Stack.Push(stack1, 30);

Stack.Create(stack2, 3); // -> crea pila con una capacità di 3 elementi
Stack.Push(stack2, -10);
Stack.Push(stack2, -20);

Visualizza(stack1); //-> 30, 20, 10
Visualizza(stack2); //-> -20, -10
}

static void Visualizza(StackData stack)
{
    while (Stack.IsEmpty(stack) == false)
    {
        Console.WriteLine(Stack.Pop(stack));
    }
}

```

### 3.1 Vantaggi e limiti dell'uso di modulo+record

Questa soluzione supera il limite fondamentale visto in precedenza, ma la divisione in due unità di codice, record e modulo, presenta diverse problematiche:

- Non è una soluzione completamente incapsulata: i campi di `StackData`, ovviamente pubblici, sono accessibili ovunque nel programma; ciò rende "trasparente" l'implementazione della pila.
- Due unità complicano l'organizzazione e il riutilizzo del codice, essendo doppio il numero di componenti da gestire. (Si pensi a una soluzione analoga anche per lista, coda, dizionario, etc.)
- Il codice esterno dipende da due componenti, `StackData` e `Stack`, e non da uno soltanto.
- La creazione della pila si svolge in due fasi: creazione della variabile record ed esecuzione del metodo `Create()`.
- In generale: l'uso della pila non è particolarmente *user friendly*, a partire dall'invocazione dei metodi, che risulta piuttosto "spigolosa".

(Nota a margine: nel modello procedurale, questi limiti sono superabili utilizzando una tecnica che adotta il concetto di *handle* (maniglia); questo rappresenta una chiave (di solito un intero) che identifica l'oggetto da elaborare e che viene passato ai metodi del modulo per ogni operazione da eseguire.)

## 4 Programmazione OO col modello procedurale

Il modello di programmazione *object oriented* nasce per risolvere i problemi precedentemente elencati; prima di introdurlo compiutamente, però, intendo fornire un'ulteriore soluzione basata sul modello procedurale. Lo scopo, come sarà evidente più avanti, è mostrare che il modello OO non è altro che il modello procedurale con qualche "ritocco" sintattico.

### 4.1 "Fondere" modulo e record

È possibile risolvere gran parte dei problemi elencati in (3.1) definendo un'unità di codice "ibrida", che riunisca il record `StackData` e il modulo `Stack`:

```
class Stack
{
    private int[] items;
    private int head;

    public static void Create(Stack stack, int capacity)
    {
        stack.items = new int[capacity];
        stack.head = -1;
    }

    public static void Push(Stack stack, int item)
    {
        stack.head++;
        stack.items[stack.head] = item;
    }

    public static int Pop(Stack stack)
    {
        return stack.items[stack.head--];
    }

    public static bool IsEmpty(Stack stack)
    {
        return stack.head == -1;
    }

    public static bool IsFull(Stack stack)
    {
        return stack.head == stack.items.Length-1;
    }
}
```

Nota bene: i campi non sono statici e sono privati, dunque inaccessibili al codice esterno. I metodi non accedono direttamente ai campi; lo fanno sempre attraverso un parametro, in questo caso di tipo `Stack`.

Il frammento di codice che segue mostra che, nell'uso, non è cambiato molto rispetto al capitolo precedente:

```

static void Main(string[] args)
{
    Stack stack1 = new Stack();
    Stack stack2 = new Stack();

    Stack.Create(stack1, 5); // -> crea pila con una capacità di 5 elementi
    Stack.Push(stack1, 10);
    Stack.Push(stack1, 20);
    Stack.Push(stack1, 30);

    Stack.Create(stack2, 3); // -> crea pila con una capacità di 3 elementi
    Stack.Push(stack2, -10);
    Stack.Push(stack2, -20);

    Visualizza(stack1); //-> 30, 20, 10
    Visualizza(stack2); //-> -20, -10
}

static void Visualizza(Stack stack)
{
    while (Stack.IsEmpty(stack) == false)
    {
        Console.WriteLine(Stack.Pop(stack));
    }
}

```

Adesso esiste un solo componente, che è totalmente incapsulato, poiché soltanto i suoi metodi possono accedere ai campi, anche se indirettamente attraverso un parametro.

## 4.2 Conclusioni

Il costrutto appena realizzato può essere considerato un *oggetto* a tutti gli effetti. Con questa tecnica potremmo implementare non solo liste, code, dizionari, ma anche *button*, *textbox*, *listbox*, e qualsiasi altro tipo di oggetto attualmente disponibile.

A questo modello ibrido di programmazione manca soltanto una sintassi adeguata, che semplifichi l'implementazione e l'uso degli oggetti.

## 5 Modello di programmazione object oriented

I linguaggi *object oriented* forniscono una sintassi specifica per definire e usare gli *oggetti*, i quali, "dietro le quinte" sono implementati come in (4.1). Nella terminologia OO un *oggetto* è definito *classe*. Segue la classe `Stack`:

```
class Stack
{
    private int[] items;
    private int head;

    public Stack(int capacity) // costruttore: sostituisce Create()
    {
        items = new int[capacity];
        head = -1;
    }

    public void Push(int item)
    {
        head++;
        items[stack.head] = item;
    }

    public int Pop()
    {
        return items[head--];
    }

    public static bool IsEmpty()
    {
        return head == -1;
    }

    public bool IsFull()
    {
        return head == items.Length-1;
    }
}
```

Ci sono due cose degne di nota:

- Il metodo di creazione, `Create()`, è stato sostituito da un metodo speciale, il *costruttore*; ha lo stesso nome della classe e viene invocato all'atto della creazione dell'oggetto.
- I metodi non sono più statici e hanno apparentemente perso il primo (o unico) parametro; internamente accedono direttamente ai campi della classe. Nella terminologia OO, i metodi vengono definiti "**di istanza**", in contrapposizione ai metodi statici. La parola *istanza* (della classe) designa l'oggetto, la variabile, attraverso il quale sono invocati i metodi.

È importante comprendere che questo cambiamento nella sintassi non modifica il modello di programmazione sottostante, equivalente a quello procedurale. Di seguito mostro il costruttore e

il metodo `Push()`, usando entrambi i tipi di sintassi:

#### Sintassi *object oriented*

```
public Stack(int capacity)
{
    items = new int[capacity];
    head = -1;
}
```

```
public int Pop()
{
    return items[head--];
}
```

#### Sintassi procedurale

```
public static void Create(Stack stack, int capacity)
{
    stack.items = new int[capacity];
    stack.head = -1;
}
```

```
public static int Pop(Stack stack)
{
    return stack.items[stack.head--];
}
```

Nel codice procedurale ho evidenziato gli elementi aggiuntivi rispetto al codice OO; si tratta di elementi aggiunti automaticamente dal linguaggio. (I nomi del parametro `stack` e del metodo `Create()` sono stati decisi dal sottoscritto.)

## 5.1 Uso degli oggetti

Con la sintassi *object oriented* scompare completamente il modulo e assume centralità l'oggetto, e cioè l'istanza (le variabili `stack1` e `stack2`, nell'esempio):

```
static void Main(string[] args)
{
    Stack stack1 = new Stack(5);
    stack1.Push(10);
    stack1.Push(20);
    stack1.Push(30);

    Stack stack2 = new Stack(3);
    stack2.Push(-10);
    stack2.Push(-20);

    Visualizza(stack1); //-> 30, 20, 10
    Visualizza(stack2); //-> -20, -10
}
```

Ma, ancora una volta, il linguaggio trasforma la sintassi OO in quella procedurale:

#### Sintassi *object oriented*

```
Stack stack1 = new Stack(5);
```

```
stack1.Push(10);
```

#### Sintassi procedurale

```
Stack stack1 = new Stack();
Stack.Create(stack1, 5);
```

```
Stack.Push(stack1, 10);
```

## 5.2 Accesso all'oggetto nei metodi di istanza: parola chiave "this"

Nella sintassi OO i metodi accedono direttamente ai campi della classe, similmente a quanto avviene nei metodi statici di un modulo che accedono ai campi statici dello stesso. Ma si tratta di una similitudine soltanto apparente.

Non bisogna dimenticare che un metodo di istanza riceve come primo parametro un oggetto (l'istanza, appunto) della classe. Questo parametro è nascosto al programmatore, ma esiste, ed è possibile utilizzarlo esplicitamente mediante la parola chiave `this`.

Nel confronto seguente, `this` equivale al parametro `stack`:

#### Sintassi *object oriented*

```
public int Pop()
{
    return this.items[this.head--];
}
```

#### Sintassi *procedurale*

```
public static void Pop(Stack stack)
{
    return stack.items[stack.head--];
}
```

Dunque, `this` riferenzia l'oggetto attraverso il quale viene chiamato il metodo. Ad esempio:

```
stack1.Push(10); //-> dentro il metodo Push(), "this" riferenzia "stack1" (istanza)
...
stack2.Push(-10); //-> dentro il metodo Push(), "this" riferenzia "stack2" (istanza)
```

## 5.3 “Status” speciale dei costruttori

Il costruttore è un metodo che viene trattato in modo speciale dal linguaggio. Innanzitutto non ha un nome proprio; internamente, il compilatore gli assegna il nome `ctor` (acronimo di *constructor*). Viene eseguito contestualmente alla creazione dell'oggetto; dopodiché, *non può essere invocato direttamente, né dal codice esterno, né dai metodi della classe*.

Il costruttore, inoltre, è l'unico metodo nel quale si può assegnare un valore ai campi *readonly*. Un campo decorato con la parola chiave `readonly` può essere assegnato una sola volta e solo nel costruttore (oppure durante la dichiarazione).

Ad esempio, supponiamo che il campo `items` sia dichiarato *readonly*:

```
class Stack
{
    private readonly int[] items;
    private int head;

    public Stack(int capacity)
    {
        items = new int[capacity]; // è possibile soltanto nel costruttore!
        head = -1;
    }
    ...
}
```

## 6 Oggetti “svelati” (in linguaggio assembly)

Nei precedenti paragrafi ho mostrato l'equivalenza tra i modelli *object oriented* e procedurale, e il modo in cui primo viene “tradotto” nel secondo. Va precisato, comunque, che il compilatore non produce nuovi costrutti in C#, ma genera codice macchina<sup>1</sup>, il quale rispetta la struttura del modello procedurale presentato in (4.1).

Di seguito mostro la corrispondenza tra i due modelli di programmazione, confrontando definizione e uso di un *oggetto* con il corrispondente linguaggio *assembly*.

### 6.1 Classe Punto

La classe `Punto` definisce un punto in un piano cartesiano di coordinate intere. La classe ha un costruttore e un metodo, `Trasla(int offset)`, che modifica la posizione di un valore stabilito:

```
class Punto
{
    public int X;
    public int Y;
    public Punto(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }

    public void Trasla(int offset)
    {
        X += offset;
        Y += offset;
    }

    public static void Trasla(Punto @this, int offset) // versione procedurale di Trasla()
    {
        @this.X += offset;
        @this.Y += offset;
    }
}
```

Ho definito anche una versione procedurale del metodo `Trasla()`, allo scopo di mostrare la sua equivalenza con la versione *object oriented*. Segue il codice che usa la classe:

```
public static void Main()
{
    Punto p = new Punto(5, 10);
    p.Trasla(10); // esegue versione object oriented
    Punto.Trasla(p, 20); // esegue versione "procedurale"
}
```

1 Per esattezza, il compilatore C# produce codice **IL** (*Intermediate Language*). All'atto dell'esecuzione, quest'ultimo viene tradotto in codice macchina da un altro compilatore, chiamato *jitter* (*Just In Time Compiler*).

## 6.2 Analisi del codice corrispondente in linguaggio *assembly*

Alcune note generali sul codice *assembly*:

- I registri `ecx` e `esi` sono usati per memorizzare indirizzi di memoria.
- Quando un metodo restituisce un valore intero o un indirizzo, lo memorizza in `eax`.
- Passaggio degli argomenti: il primo argomento viene passato mediante il registro `edx`; dal secondo in poi viene usata la memoria *stack* e dunque l'istruzione `push`.

Creazione dell'oggetto	
	<code>mov ecx, 0x20c51750</code>
	<code>call 0x5c330c8</code> 'alloca memoria
	<code>mov esi, eax</code> '-> reference "p"
Punto p = new Punto(5, 10);	<code>push 0xa</code> '-> 10
	<code>mov ecx, esi</code> '-> reference "p"
	<code>mov edx, 0x5</code> '-> 5
	<code>call dword [0x20c51734]</code> '-> costruttore

In arancione ho evidenziato la chiamata alla funzione di sistema che alloca la memoria per l'oggetto. Questa restituisce l'indirizzo dell'oggetto in `eax`; quindi viene chiamato il costruttore (in verde). Prima della chiamata del costruttore:

- In `ecx` viene memorizzato l'indirizzo di `p`.
- In `edx` viene memorizzato il primo argomento (5).
- Nello *stack* viene memorizzato il secondo argomento (10)

In sostanza, al costruttore sono passati tre parametri, il primo dei quali è l'indirizzo dell'oggetto.

Esecuzione del metodo: <code>Trasla(int offset)</code> (metodo di istanza)	
p.Trasla(10);	<code>mov ecx, esi</code> '-> reference "p"
	<code>mov edx, 0xa</code> '-> 10
	<code>call dword[0x20f61740]</code> '-> Trasla()

Nota bene: prima di invocare il metodo, in `ecx` viene memorizzato l'indirizzo dell'oggetto.

Esecuzione del metodo: <code>Trasla(Punto @this, int offset)</code> (metodo statico)	
Punto.Trasla(p, 10);	<code>mov ecx, esi</code> '-> reference "p"
	<code>mov edx, 0xa</code> '-> 10
	<code>call dword[0x20f6174c]</code> '-> Trasla()

Il codice *assembly* è perfettamente identico a quello della versione *object oriented* e coerente con i parametri passati al metodo (*reference* oggetto e valore 10)



## 6.2.1 Codice assembly corrispondente alla classe Punto

Alcune note generali sul codice *assembly*:

- I due campi della classe, `X` e `Y`, occupano 4 byte ciascuno e si trovano agli indirizzi:
- `X` → indirizzo oggetto + 4.
- `Y` → indirizzo oggetto + 8.



Ho colorato in grigio le istruzioni che non sono rilevanti.

Costruttore	
<pre>public Punto(int x, int y) {     this.X = x;     this.Y = y; }</pre>	<pre>push ebp mov ebp, esp mov [ecx + 0x4], edx    'X = x (5) mov eax, [ebp + 0x8]    'eax = y (10) mov [ecx + 0x8], eax    'Y = eax (10) pop ebp ret 0x4</pre>

`ecx` memorizza l'indirizzo dell'oggetto. Il codice copia i due parametri nelle locazioni di memoria riservate a X (`ecx + 4`) e Y (`ecx + 8`).

(L'assegnazione di Y richiede due istruzioni perché il parametro si trova nello *stack* (`ebp + 8`) e l'istruzione `mov` non può eseguire un accesso simultaneo a due locazioni di memoria.

Metodo Trasla()	
<i>object oriented</i>	
<pre>public void Trasla(int offset) {     X += offset;     Y += offset; }</pre>	<pre>push ebp mov ebp, esp add [ecx + 0x4], edx    'X += offset add [ecx + 0x8], edx    'Y += offset pop ebp ret</pre>
<i>procedurale</i>	
<pre>public void Trasla(Punto @this, int offset) {     @this.X += offset;     @this.Y += offset; }</pre>	<pre>push ebp mov ebp, esp add [ecx + 0x4], edx    'X += offset add [ecx + 0x8], edx    'Y += offset pop ebp ret</pre>

In *assembly*, i due metodi sono identici.

## 6.3 Conclusioni

L'esempio precedente rappresenta una "radiografia" del modello di programmazione *object oriented*, e mostra che questo è basato sul modello procedurale. Analizzando il codice *assembly* si comprende che:

- I metodi di istanza ricevono come primo argomento l'oggetto attraverso il quale sono chiamati (*l'istanza*, appunto). L'indirizzo dell'oggetto è memorizzato in `ecx` e viene utilizzato nel metodo per accedere ai suoi campi.
- È il compilatore a produrre questa "traduzione" dei metodi di istanza in metodi statici che dichiarano un parametro nascosto.
- Il costruttore è un metodo a tutti gli effetti; anch'esso riceve l'indirizzo dell'oggetto, dopo che questo è già stato creato in memoria.

In sostanza, questa analisi mostra l'equivalenza tra la classe definita in (5) e il costrutto definito in (4.1); le differenze si riducono alla sintassi e a un trattamento speciale che il C# riserva al costruttore.