Oggetti “undercover”

Implementazione del modello object oriented

Anno 2019/2020

Indice generale

1 Introduzione: (*programmazione procedurale*)

Questo tutorial ha lo scopo di mostrare che il modello di programmazione *object oriented* è basato sul modello di programmazione procedurale, con l'aggiunta di una specifica sintassi. Di fatto sarebbe possibile programmare in modo "object oriented" anche utilizzando il solo modello procedurale, però in modo poco elegante e complicato.

Per mostrare l'equivalenza dei due modelli, userò un approccio simile a quello impiegato nel tutorial **Oggetti**: partirò dal presupposto che il modello *object oriented* non esista e affronterò un semplice esempio utilizzando il modello procedurale limitato all'uso dei *moduli*. Quindi:

- 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 è il modello di programmazione *object oriented* adottato dal C#.

2 Implementare una “pila” (col modello procedurale)

Supponi, 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) //-> IndexOutOfRange se la pila è piena
    {
        head++;
        items[head] = item;
    }

    public static int Pop() //-> IndexOutOfRange 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 Limiti dell'uso di un *modulo*

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

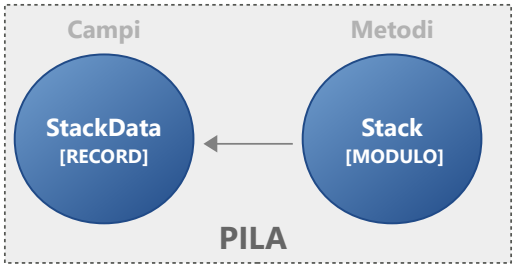
- Uso di una funzione senza dipendere dalla sua implementazione.
- Riutilizzo di una funzione in altri contesti / applicazioni.

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

Si tratta di un limite inaccettabile per una pila, come per oggetti simili. (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`. Occorre dunque separare i dati dai metodi che li elaborano, gestendo i primi mediante un *record* e i secondi mediante un *modulo*:




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

Ogni metodo del *modulo* dichiarerà come primo parametro uno `StackData` ed eseguirà su di esso l'operazione richiesta:

```
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 mostra l'uso 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*

Rappresentare il concetto di pila mediante due componenti, *record+modulo,* supera il limite visto in precedenza, ma la divisione in due unità di codice presenta delle problematiche:

- Non è una soluzione completamente incapsulata: i campi di `StackData` 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 dell'istanza del *record* ed esecuzione del metodo `Create()`, che inizializza i campi della pila.
- In generale: l'uso della pila non è particolarmente *user friendly*, a partire dall'invocazione dei metodi, che risulta piuttosto "spigolosa".

4 Programmazione OO col modello procedurale

Il modello di programmazione *object oriented* nasce per risolvere i problemi precedentemente elencati; prima di mostrarlo compiutamente, però, intendo fornire un'ulteriore soluzione basata sul modello procedurale. Lo scopo è quello di 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'unita di codice 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 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 *oggetto* attualmente disponibile.

A un simile modello 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). 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'istanza.
- 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* designa l'oggetto attraverso il quale sono invocati.

È fondamentale 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 apacity)
{
    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.

In sintesi, il compilatore traduce il codice di sinistra in quello di destra.

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'istanza 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` referenzia l'oggetto attraverso il quale viene chiamato il metodo. Ad esempio:

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

5.4 Vincoli sui membri statici negli *oggetti*

Nel tutorial **Oggetti** ho mostrato che gli *oggetti* possono definire membri statici, sia campi che metodi, e che questi possono servire a svariati scopi. Ho anche affermato, senza fornire particolari spiegazioni, che *i metodi statici non possono utilizzare i membri d'istanza*, siano essi campi o altri metodi. Qui approfondisco la questione e, utilizzando il modello di programmazione introdotto in 4, dimostrerò che questo vincolo è un'inevitabile conseguenza del modello OO.

Segue la classe `Punto`, che rappresenta un punto nel piano. La classe definisce due metodi che modificano la posizione del punto; il primo è un metodo d'istanza, il secondo è statico[1]:

```
class Punto
{
    public int X; // ho definito pubblici i campi per semplificare il codice
    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(int offset)
    {
        X += offset;  // Errore: non esiste l'istanza che memorizza X e Y
        Y += offset;  // ...
    }
}
```

Il metodo statico viola le regole del linguaggio, ed è semplice comprendere perché se si trasforma il modello OO in quello procedurale – del tutto equivalente – introdotto in 4.

```
class Punto
{
    public int X;
    public int Y;

    public static void Create(Punto @this, int x, int y)
    {
        @this.X = x;
        @this.Y = y;
    }
```

1 Infrango le regole di buona progettazione, definendo dei campi pubblici, allo scopo di rendere più semplice il codice la successiva dimostrazione.

```
    public static void Trasla(Punto @this, int offset)  // ex metodo d'istanza
    {
        @this.X += offset;
        @this.Y += offset;
    }

    public static void Trasla(int offset)
    {
        X += offset;  // a chi appartengono i campi X e Y?
        Y += offset;
    }
}
```

Il metodo d'istanza viene tradotto in un metodo statico che dichiara come primo parametro un'istanza "nascosta" della classe (`@this`). Ma un metodo statico non subisce alcuna traduzione da parte del linguaggio, e dunque non dichiara il parametro nascosto a cui dovrebbero appartenere i campi utilizzati nel codice.

Considera adesso l'esecuzione dei due metodi, supposto di aver già creato un'istanza:

```
Punto p = new Punto(5, 10);
```

La chiamata del metodo d'istanza viene tradotta in:

```
p.Trasla(3);                                -> Punto.Trasla(p,3);
```

Dunque, l'istanza `p` contiene i campi `X` e `Y` utilizzati nel metodo.

Ma la chiamata al metodo statico non subisce alcuna traduzione:

```
Punto.Trasla(3);                              -> Punto.Trasla(3);
```

Dunque: nell'esecuzione del metodo non è coinvolta nessuna istanza, dunque non è possibile elaborare i campi `X` e `Y`, poiché, semplicemente, non esistono!

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[2], il quale rispetta la struttura del modello procedurale presentato in (4.1).

Di seguito dimostro la corrispondenza tra i due modelli di programmazione confrontando la definizione e l’uso di un *oggetto* con il corrispondente linguaggio *assembly*. Per farlo, riparto dalla classe `Punto` introdotta nel paragrafo precedente, con qualche modifica:

```
class Punto
{
    public int X;  // ho definito pubblici i campi per semplificare il codice
    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 statica di Trasla()
    {
        @this.X += offset;
        @this.Y += offset;
    }
}
```

In questo versione, il metodo statico `Trasla()` non viola alcuna regola, poiché riceve un’istanza passata come primo parametro. Lo scopo è quello di mostrare che le due versioni del metodo – statica e di istanza – sono perfettamente equivalenti.

Segue un frammento di codice che usa la classe:

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

2 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*).

6.1 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

```
Punto p = new Punto(5, 10);
```

```
mov  ecx, 0x20c51750
call 0x5c330c8          'alloca memoria
mov  esi, eax           '-> reference "p"
push 0xa                '-> 10 in stack
mov  ecx, esi           '-> reference "p"
mov  edx, 0x5           '-> 5
call dword [0x20c51734] '-> costruttore
```

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

- In `exc` 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 `p`.

Esecuzione del metodo: Trasla(int offset) (metodo di istanza)

```
p.Trasla(10);
```

```
mov  ecx, esi           '-> reference "p"
mov  edx, 0xa           '-> 10
call dword[0x20f61740]  '-> Trasla()
```

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

Esecuzione del metodo: Trasla(Punto @this, int offset) (metodo statico)

```
Punto.Trasla(p, 10);
```

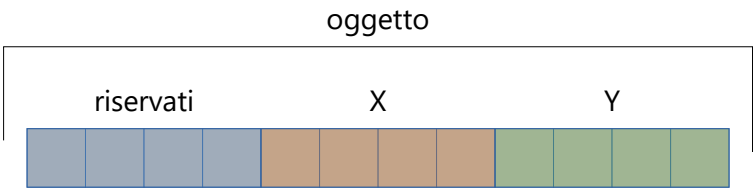
```
mov  ecx, esi           '-> reference "p"
mov  edx, 0xa           '-> 10
call dword[0x20f6174c]  '-> Trasla()
```

Nota bene: il codice *assembly* che esegue i due metodi è lo stesso in entrambi i casi! Viene passato come primo argomento l'indirizzo dell'istanza, come secondo argomento il valore 10.

6.1.1 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 istanza + 4.
- `Y` → indirizzo istanza + 8.



Ho colorato in grigio le istruzioni che non sono rilevanti.

Costruttore

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

```
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
```

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

object oriented

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

```
push ebp
mov ebp, esp
add [ecx + 0x4], edx     'X += offset
add [ecx + 0x8], edx     'Y += offset
pop ebp
ret
```

procedurale

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

```
push ebp
mov ebp, esp
add [ecx + 0x4], edx     'X += offset
add [ecx + 0x8], edx     'Y += offset
pop ebp
ret
```

In *assembly*, i due metodi, d'istanza e statico, sono identici!

6.2 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'indirizzo dell'oggetto attraverso il quale sono chiamati (l'*istanza*, appunto). L'indirizzo 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 primo 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.