

Esempi di polimorfismo

Scenari d'implementazione dei tipi astratti predefiniti

Compatibilità: dotNET 2.0+

Anno 2017/2018

Indice generale

| | | |
|----------|--|-----------|
| 1 | Introduzione..... | 3 |
| 2 | Formattazione in stringa: <code>object.ToString()</code>..... | 4 |
| 2.1 | Tipo <code>Object</code> | 5 |
| 2.2 | Funzionamento del metodo <code>string.Format()</code> | 6 |
| 2.3 | Rendere un tipo “Formattabile”..... | 6 |
| 3 | Formattazione personalizzata: <code>IFormattable</code>..... | 7 |
| 3.1 | Caratteristica della “formattabilità”..... | 7 |
| 3.2 | Interfaccia <code>IFormattable</code> | 8 |
| 3.2.1 | Tipo <code>TimeRange</code> | 8 |
| 3.3 | Uso dell’interfaccia nel metodo <code>Format()</code> | 9 |
| 3.4 | Conclusioni..... | 10 |
| 4 | Concetto di sequenza: interfaccia <code>IEnumerable</code>..... | 11 |
| 4.1 | Elaborare sequenze di oggetti: usare <code>IEnumerable</code> | 12 |
| 4.2 | Implementare l’interfaccia <code>IEnumerable</code> | 12 |
| 4.3 | Conclusioni..... | 14 |

1 Introduzione

In questo tutorial propongo alcuni esempi di programmazione che prevedono l'uso di tipi già definiti in .NET: interfacce, classi astratte, classi concrete con metodi virtuali. L'obiettivo è quello di affrontare scenari che mostrino l'importanza dei tipi astratti in generale e come questi siano integrati, insieme ai tipi concreti, all'interno del .NET Framework.

Ogni esempio proposto avrà come obiettivo la realizzazione di una classe che implementa (o deriva) uno o più tipi astratti allo scopo di poter essere utilizzabile in determinati scenari. Prenderò in considerazione i seguenti tipi:

- Classe (concreta) `Object`.
- Interfaccia `IFormattable`.
- Interfaccia `IEnumerable`.
- Classe (astratta) `Stream`.
- Classe (concreta) `Collection`.

2 Formattazione in stringa: `object.ToString()`

Considera il seguente codice:

```
double n = 12.5;
DateTime adesso = DateTime.Now;
string s = string.Format("[{0}] [{1}]", n, adesso);
Console.WriteLine(s);
```

che produce il seguente output:

```
[12,5] [05/03/2018 21:44:41]
```

Tutta l'azione si svolge nel metodo `Format()` che, evidentemente, è in grado di ottenere una stringa da ognuno dei parametri. La questione è come ci riesca. Non si tratta di una questione banale; infatti, nel seguente caso l'output non è quello che ci aspetteremmo:

```
class Persona
{
    public string Nome;
    public string Cognome;
}
...
var p = new Persona { Nome = "Pippo", Cognome = "Spada"};
DateTime adesso = DateTime.Now;
var s = string.Format("[{0}] [{1}]", p, adesso);
Console.WriteLine(s);
```

```
[Polimorfismo.ConsoleUI.Persona] [05/03/2018 21:46:32]
```

Dunque, `Format()` non è in grado di ottenere una stringa significativa da qualsiasi parametro; dipende dal tipo del parametro. La questione si complica se si considera la *signature* del metodo¹:

```
public static string Format(string format, object arg0, object arg1);
```

Ricapitolando:

1. I parametri sono di tipo `object`, eppure `Format()` è in grado di elaborare qualsiasi tipo di dato.
2. Per ogni parametro, `Format()` ottiene una stringa, ma soltanto (apparentemente) per i tipi predefiniti è una stringa che rappresenta il valore del parametro.

Entrambe le questioni richiedono una spiegazione, la cui radice risiede nella definizione e nel ruolo del tipo `Object`.

¹ In realtà esistono 8 versioni del metodo.

2.1 Tipo Object

`Object` (alias `object`) è il tipo fondamentale dell'intero *type system* di .NET; infatti, qualsiasi tipo di dato deriva da `Object` e dunque ne eredita i metodi. Segue lo scheletro della classe (non riporto i metodi statici):

```
public class Object
{
    public Object() { }
    public virtual bool Equals(Object obj) { }
    public virtual int GetHashCode() { }
    public Type GetType() { }

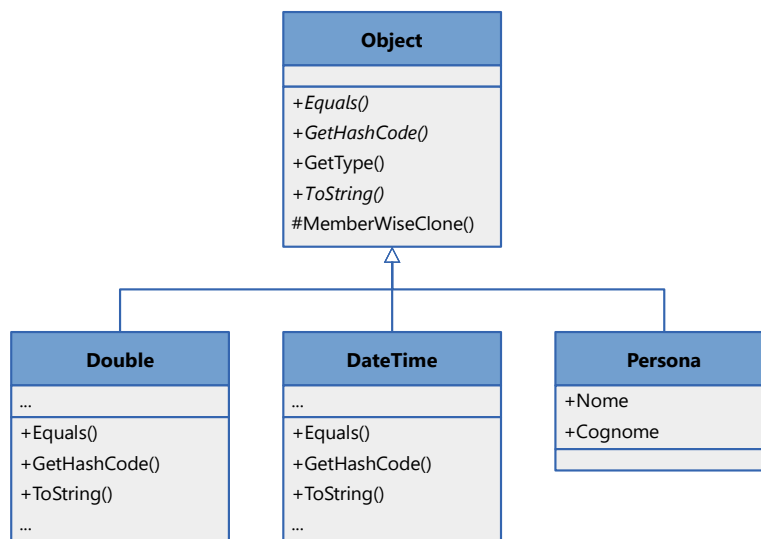
    public virtual string ToString()
    {
        return GetType().ToString();
    }

    protected Object MemberwiseClone() { }
}
```

Nota bene: ho evidenziato i metodi virtuali e riportato l'implementazione del metodo `ToString()`. Quest'ultima visualizza il nome, completo di *namespace*, del tipo effettivo dell'oggetto, come mostra il seguente frammento di codice che usa `GetType()`:

```
var persona = new Persona { Nominativo = "Pippo, Spada" };
DateTime adesso = DateTime.Now;
string nomeTipoAdesso = adesso.GetType().ToString(); //-> "System.DateTime"
string nomeTipoPersona = persona.GetType().ToString(); //-> "Polimorfismo.ConsoleUI.Persona"
```

Segue lo schema UML che mostra la relazione tra `double`, `DateTime`, `Persona` e `Object`:



Nota bene: `double` e `DateTime` ridefiniscono i metodi virtuali di `Object`, mentre `Persona` si limita a ereditarli.

2.2 Funzionamento del metodo `string.Format()`

Segue uno pseudo codice super semplificato del metodo `Format()`².

```
static string Format(string format, object arg0, object arg1)
{
    // per ogni carattere di "format"
    // se trova un segnaposto
    // formatta argomento corrispondente: (arg0 o arg1).ToString()
    // e aggiungi risultato alla stringa finale
    // ...
}
```

La parte evidenziata fornisce la spiegazione sul funzionamento del metodo: viene eseguito `ToString()` sul parametro corrispondente al segnaposto.

In conclusione, `Format()` ignora completamente il tipo effettivo degli argomenti; si limita a invocare `ToString()` su di essi. La versione di `ToString()` effettivamente eseguita dipende dal fatto che il tipo dell'argomento fornisca una propria versione oppure erediti quella definita in `Object`.

2.3 Rendere un tipo “Formattabile”

Premesso che non ha senso implementare per ogni tipo la conversione in stringa, per farlo è sufficiente ridefinire il metodo `ToString()`:

```
class Persona
{
    public string Nome;
    public string Cognome;

    public override string ToString()
    {
        return $"{Cognome}, {Nome}";
    }
}
```

Dopo questa modifica, l'output rispetta le attese, poiché il metodo eseguito sugli oggetti di tipo `Persona` non è più quello ereditato da `Object`, ma quello definito nella classe:

```
[Spada, Pippo] [05/03/2018 21:46:32]
```

2 Il codice reale è piuttosto lungo e complicato, e coinvolge diversi metodi.

3 Formattazione personalizzata: IFormattable

Considera nuovamente il codice proposto all’inizio del precedente capitolo, con qualche piccola modifica:

```
double n = 12.5;
DateTime adesso = DateTime.Now;
string s = string.Format("[{0:F2}] [{1:d}] [{1:D}]", n, adesso);
Console.WriteLine(s);
```

Ecco l’output, che è chiaramente diverso da quello prodotto nel primo capitolo:

```
[12,50] [06/03/2018] [martedì 6 marzo 2018]
```

Questo è regolato dalle *stringhe di formato* (`F2`, `d`, `D`) collocate nei segnaposti, le quali specificano la formattazione da applicare ai parametri. Ebbene, la spiegazione fornita nel precedente capitolo non può giustificare questo risultato, poiché `ToString()` non accetta parametri e dunque non può produrre una stringa diversa in base a un fattore esterno.

Infatti, `Format()` funziona secondo un meccanismo leggermente diverso da quello presentato.

3.1 Caratteristica della “formattabilità”

Mettiamoci nei panni dei programmatori Microsoft e consideriamo il seguente problema: vogliamo dare la possibilità di creare dei tipi di dati capaci di produrre una stringa in base a un parametro fornito dall’esterno (la *stringa di formato*). Il metodo virtuale `ToString()` definito in `Object` non va bene, poiché non accetta parametri. Ma possiamo aggirare il problema:

- Modifichiamo `ToString()`, aggiungendo un parametro. Si tratta di una soluzione problematica, poiché in molti scenari non esiste la necessità di specificare una stringa di formato, ma saremmo costretti a farlo. Ad esempio, saremmo costretti a scrivere:

```
lblNumeroOrdini.Text = numOrdini.ToString(null);
```

- Definire due metodi `ToString()`, di cui uno con parametro. A questo punto, però, tutti i tipi erediterebbero due metodi `ToString()`, di cui uno probabilmente inutile.³

La questione centrale è: soltanto per alcuni tipi è significativo implementare una formattazione personalizzabile. In sostanza è utile introdurre il concetto astratto di *formattabilità*, cioè un *tratto distintivo che caratterizza i tipi in grado di produrre una conversione in stringa personalizzabile*.

3 Per inciso è ciò che effettivamente accade per tutti i metodi virtuali di `Object`, e alcuni ingegneri Microsoft sostengono che non sia stata una buona scelta definire in `Object` tali metodi, poiché la maggiore parte dei tipi non ne hanno bisogno.

3.2 Interfaccia IFormattable

L'interfaccia `IFormattable` definisce il concetto astratto di *formattabilità*. Un tipo che implementa questa interfaccia è in grado di fornire una rappresentazione stringa personalizzabile.

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

L'unico metodo, `ToString()`, accetta come primo parametro una *stringa di formato* (possiamo ignorare il secondo parametro). È compito dei tipi che implementano l'interfaccia utilizzare il parametro per stabilire il tipo di formattazione da applicare.

3.2.1 Tipo TimeRange

Il tipo `TimeRange` implementa un intervallo di tempo (vedi tutorial **TimeRange**) ed è un buon candidato per mostrare l'applicazione del concetto di *formattabilità*. L'attuale versione ridefinisce già il metodo `ToString()` ereditato da `Object`:

```
public struct TimeRange
{
    ...
    public override string ToString()
    {
        string segno = (Ore < 0 || Minuti < 0 || Secondi < 0) ? "-" : "";
        return string.Format("{0}{1}:{2}:{3}", segno, Math.Abs(Ore), Math.Abs(Minuti),
                               Math.Abs(Secondi));
    }
}
```

Ciò consente di scrivere codice come il seguente:

```
var tr = new TimeRange(1, 2, 3); // (1 ora, 2 minuti e 3 secondi)
Console.WriteLine(tr);
```

e ottenere l'output atteso:

```
1:2:3
```

Desidero aggiungere a `TimeRange` la possibilità di utilizzare un diverso formato, che, nell'esempio, produrrebbe il seguente output⁴: **1 ore 2 minuti 3 secondi**.

Il formato è specificato dalla lettera `T`; pertanto mi aspetto che il codice:

```
var tr = new TimeRange(1, 2, 3); //->1:2:3
var s = string.Format("[{0}] [{0:T}]", tr);
Console.WriteLine(s);
```

produca l'output:

4 Formato senz'altro migliorabile, ma rappresenta soltanto un esempio per introdurre un concetto.


```
[1:2:3] [1 ore 2 minuti 3 secondi]
```

Ecco la nuova versione di `TimeRange`:

```
public struct TimeRange: IFormattable
{
    ...
    public override string ToString() {...}

    public string ToString(string format, IFormatProvider formatProvider)
    {
        if (format == "T")
        {
            string segno = (Ore < 0 || Minuti < 0 || Secondi < 0) ? "-" : "";
            return string.Format("{0}{1} ore {2} minuti {3} secondi", segno, Math.Abs(Ore),
                                Math.Abs(Minuti), Math.Abs(Secondi));
        }
        return ToString(); //qualunque altro formato produce l'output standard!
    }
}
```

3.3 Uso dell'interfaccia nel metodo Format()

Il tratto della *formattabilità* ha una caratteristica importante: è opzionale. `Format()` può invocare `ToString()` per ogni argomento, poiché qualsiasi tipo definisce questo metodo (lo eredita da `Object`); ma non può fare altrettanto con `ToString(string format...)`, poiché soltanto i tipi che implementano `IFormattable` lo definiscono.

Occorre dunque sapere se un oggetto appartiene a un tipo che implementa `IFormattable`. Ecco una nuova versione (in pseudo codice) di `Format()`:

```
static string Format(string format, object arg0, object arg1)
{
    // per ogni carattere di "format"
    // se trova un segnaposto
    string formatString = <estrae stringa di formato da segnaposto>;
    object arg = <seleziona arg0 o arg1>;
    string ris = FormatArg(formatString, arg)
    // aggiungi risultato alla stringa finale
    // ...
}
```

Dopo aver estratto l'eventuale stringa di formato e aver selezionato il parametro corrispondente al segnaposto, il lavoro di formattazione viene eseguito da `FormatArg()`:

```
static string FormarArg(string formatString, object arg)
{
    IFormattable fmt = arg as IFormattable;
    if (fmt != null) // il tipo run-time di arg implementa IFormattable?
        return fmt.ToString(formatString, null);
}
```

```

    return arg.ToString();
}

```

Le prime due righe sono fondamentali: l'operatore di cast `as` converte `arg` nel tipo `IFormattable`. Se l'operazione fallisce (il tipo effettivo di `arg` non implementa l'interfaccia), la variabile `fmt` viene inizializzata a `null`. Se l'operazione ha successo, viene invocato il metodo `ToString(string format...)`, altrimenti il metodo `ToString()` ereditato (ed eventualmente ridefinito) da `Object`.

Nota bene, in tutto questo il valore della *stringa di formato* non è rilevante: se `arg` implementa `IFormattable` sarà comunque invocato il metodo dell'interfaccia.

3.4 Conclusioni

La definizione in `Object` del metodo virtuale `ToString()` e, soprattutto, l'esistenza dell'interfaccia `IFormattable` mostrano in azione il fondamentale risultato prodotto dell'uso di astrazioni: *poter scrivere un procedimento che non dipende dal tipo effettivo degli oggetti che elabora*.

Il metodo `Format()` produce una stringa sulla base del valore di alcuni argomenti; *ma non conosce il tipo degli argomenti*, né implementa un procedimento per convertire un argomento in stringa. D'altra parte, `Format()` "sa":

- Che ogni argomento definisce il metodo `ToString()`, poiché qualunque tipo di dato deriva da `Object`.
- Che un argomento può implementare l'interfaccia `IFormattable`, e dunque definire il metodo `ToString(string format...)`.

Sulla base di questi due punti è stato possibile scrivere del codice in grado di formattare in stringa qualsiasi tipo di oggetto. Ciò rende possibile implementare tipi convertibili in stringa e che abbiano la caratteristica della *formattabilità*.

Naturalmente, l'intera questione non riguarda soltanto `Format()`, ma l'intero "ecosistema" che si basa sulla conversione in stringa degli oggetti.

```

var tr = new TimeRange(1, 2, 3);           // (1 ora, 2 minuti, 3 secondi)
Console.WriteLine(tr);                    //->1:2:3
Console.WriteLine("{0}", tr);              //->1:2:3
Console.WriteLine("{0:T}", tr);            //->1 ore  2 minuti  3 secondi
var s = $"{tr} [{tr:T}]";                  // (1 ora, 2 minuti, 3 secondi)
Console.WriteLine(s);                      //->[1:2:3] [1 ore  2 minuti  3 secondi]

```

```

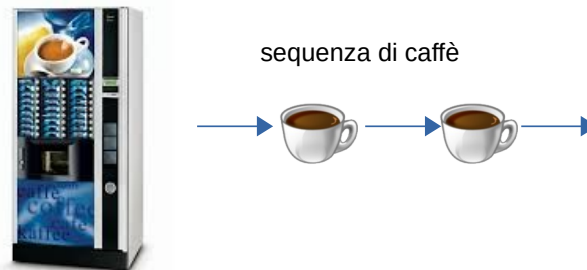
1:2:3
1:2:3
1 ore 2 minuti 3 secondi
[1:2:3] [1 ore 2 minuti 3 secondi]

```

4 Concetto di sequenza: interfaccia IEnumerable

In molti scenari il codice deve rispondere alla necessità di processare una sequenza di oggetti. Per comprendere l'importanza di questi scenari occorre innanzitutto comprendere la differenza tra i concetti di "sequenza" e "collezione", che spesso vengono usati in modo intercambiabile. Per farlo ricorrerò a una metafora.

Considera il distributore di caffè:



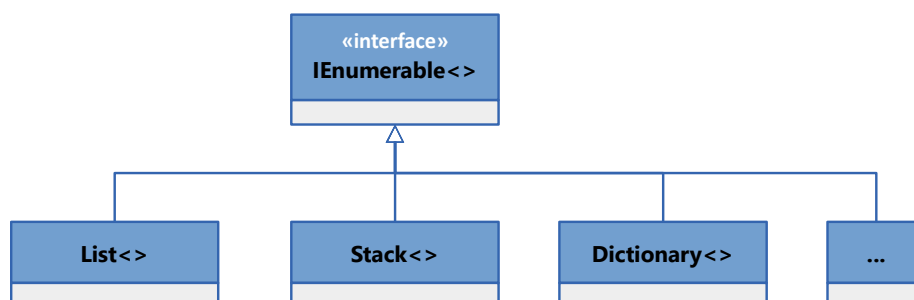
È in grado di produrre una *sequenza di caffè*, ma tutti noi sappiamo che dentro non nasconde una collezione di caffè già pronti: questi vengono preparati, e consumati, man mano che sono richiesti. In sostanza: *la sequenza di caffè non è una cosa che esiste da qualche parte, né prima (dentro il distributore), né dopo (i caffè sono consumati, man mano che sono erogati).*

D'altra parte, possiamo "collezionare" i caffè in un vassoio per portarli agli amici (e ottenere così una *collezione di caffè*). Sappiamo, inoltre, che il distributore delle bibite, pur comportandosi come quello dei caffè (produce una *sequenza di bibite*), contiene effettivamente una *collezione di bibite*.

In conclusione, *sequenza* è un concetto più astratto di *collezione*. Il secondo suggerisce l'idea di una struttura che contiene degli oggetti e definisce un insieme di operazioni; il primo, invece, fa semplicemente riferimento alla capacità di produrre un oggetto per volta, man mano che viene richiesto.

.NET implementa il concetto di sequenza attraverso l'interfaccia generica `IEnumerable<T>`. Il termine può essere tradotto in "iterabile" e cioè: un oggetto è *iterabile* se è in grado di produrre una sequenza di oggetti, uno per volta.

Naturalmente, tutti i tipi di collezione sono *iterabili*, e cioè in grado di produrre in sequenza gli elementi che memorizzano.



Per accedere agli elementi di una sequenza (dunque, di un oggetto iterabile), occorre usare il costrutto `foreach`. Questo "scorre" gli elementi uno per volta, indipendentemente dal tipo effettivo dell'oggetto iterabile che produce la sequenza: vettore, lista, pila, dizionario, etc.

4.1 Elaborare sequenze di oggetti: usare IEnumerable

Il concetto di sequenza è indipendente dal modo con il quale i dati sono gestiti, è dunque vantaggioso scrivere codice che elabori sequenze senza dipendere dalla loro origine. Lo si fa utilizzando esplicitamente il tipo `IEnumerable`.

Il seguente metodo visualizza una sequenza di nomi:

```
static void Visualizza(IEnumerable<string> listaNomi)
{
    foreach (var nome in listaNomi)
    {
        Console.WriteLine(nome);
    }
}
```

Il metodo non dipende dal tipo effettivo di `listaNomi`, poiché questa variabile referencia un oggetto iterabile, che è possibile "scorrere" mediante un `foreach`.

`Visualizza()` è utilizzabile con qualsiasi oggetto iterabile, come mostra il seguente codice:

```
string[] nomi = { "filippo", "andrea", "sara" };

List<string> listaNomi = new List<string>(nomi);
Stack<string> pilaNomi = new Stack<string>(nomi);
Queue<string> codaNomi = new Queue<string>(nomi);

Visualizza(nomi);           // -> filippo, andrea, sara
Visualizza(listaNomi);      // -> filippo, andrea, sara
Visualizza(pilaNomi);       // -> sara, andrea, filippo
Visualizza(codaNomi);       // -> filippo, andrea, sara
```

Nota bene: gli elementi della sequenza non vengono prodotti sempre nello stesso ordine; dipende dal tipo effettivo (vettore, lista, pila o coda, etc).

4.2 Implementare l'interfaccia IEnumerable

Il meccanismo sottostante all'*iterabilità* è piuttosto complicato da comprendere appieno; qui mi limito a introdurre quanto basta per poterla implementare in classi che gestiscono i dati in una collezione.

Segue un frammento della classe `AlbumFoto`, che gestisce un elenco di fotografie:

```
public class AlbumFoto
{
    List<Foto> fotoList = new List<Foto>();
    ...
    public AlbumFoto(string folder)
    {
        ...
    }
}
```

```

    public int Count
    {
        get { return fotoList.Count; }
    }

    public Foto this[int index]
    {
        get
        {
            return fotoList[index];
        }
    }
}

```

La classe incapsula una lista e fornisce l'accesso alle foto mediante un indicizzatore. Questa implementazione ha un limite: un oggetto `AlbumFoto` non può essere processato come una sequenza di foto, nonostante di fatto lo sia.

Perché ciò sia possibile deve implementare `IEnumerable`:

```

public class AlbumFoto : IEnumerable
{
    List<Foto> fotoList = new List<Foto>();
    ...
    public IEnumerator<Foto> GetEnumerator()
    {
        return fotoList.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator() // obsoleto
    {
        throw new NotImplementedException();
    }
}

```

L'interfaccia definisce il solo metodo `GetEnumerator()`⁵, il quale restituisce un `IEnumerator`. Quest'ultimo è un *iteratore*, e cioè l'oggetto che ha la funzione di produrre la sequenza.

Comprendere il funzionamento degli *iteratori* non è semplice, e in questo caso inutile; infatti, `AlbumFoto` si limita a restituire l'*iteratore* utilizzato da `fotoList`, il quale produrrà una sequenza di foto nell'ordine con il quale sono memorizzate nella lista.

Segue un frammento di codice che sfrutta l'iterabilità di `AlbumFoto`:

```

AlbumFoto album = new AlbumFoto("Immagini");
...
void LoadGallery()
{
    pnlGallery.Controls.Clear();
}

```

5 In realtà definisce due metodi `GetEnumerator()`, ma il secondo è obsoleto e può essere ignorato (ma non cancellato).

```
foreach (Foto foto in album)
{
    var fotoView = CreateFotoView(foto);
    pnlGallery.Controls.Add(fotoView);
}
```

4.3 Conclusioni

Ci sarebbe molto da aggiungere sull'implementazione degli *iteratori*, ma qui è importante comprendere il fondamentale concetto dell'*iterabilità*, e cioè la capacità di produrre una sequenza di oggetti; fondamentale perché, in moltissimi scenari, tutto ciò che serve è processare i dati in sequenza.

`IEnumerable` rappresenta questo concetto. La sua applicazione implica comprendere due punti:

- Il codice che elabora i dati in sequenza (mediante `foreach`) dovrebbe usare il tipo `IEnumerable`; in questo modo non dipenderà da un particolare tipo concreto.
- Qualunque tipo che gestisce, o è in grado di produrre, una collezione di oggetti, dovrebbe implementare l'interfaccia `IEnumerable`.

La sua implementazione in classi che gestiscono una collezione è estremamente semplice, poiché è sufficiente, nel metodo `GetEnumerator()`, chiamare l'omologo metodo della collezione.