

Capitolo 14

Collezioni, Generics, funzioni anonime, espressioni lambda

Il concetto di elemento generico è estremamente importante in C# e .Net in generale. Esso si può applicare a classi e metodi ma trova, normalmente, il suo impiego più diffuso in congiunzione con le **collezioni**. I generics, cosiddetti nella dizione inglese, fanno la loro comparsa con la versione 2.0 del framework e, ovviamente, non sono un puro esercizio di stile; la loro introduzione permette incrementi, molto significativi, di performance e il mantenimento di una rigorosa type-safety. Secondo alcuni si tratta della più importante novità in C# 2.0 Andiamo però con ordine. Il codice che segue può sembrare del tutto normale e, in pratica, lo è senz'altro:

```
C# Esempio 14.1
1  using System.Collections;
2
3  class Program
4  {
5      public static void Main()
6      {
7          ArrayList AL = new ArrayList();
8          AL.Add(100);
9          int i = (int)AL[0];
10     }
11 }
```

Facciamo prime di tutto le presentazioni al fine di chiarire qualche dettaglio del codice precedente: In .Net abbiamo il namespace **System.Collections**; nella versione 4 del framework, come nelle precedenti, questo namespace contiene molte classi, interfacce ed anche una struttura:

| Classe | Uso |
|--------------------------------|--|
| ArrayList | In pratica un array dinamico |
| BitArray | E' un array di bit espressi in forma booleana (true = 1, false = 0) |
| CaseInsensitiveComparer | Compare due elementi ignorando le maiuscole/minuscole |
| CollectionBase | E' una classe base astratta per collezioni fortemente tipizzate |
| DictionaryBase | E' una classe base astratta per collezioni di coppie chiave-valore |
| HashTable | Coppie chiave-valore organizzate tramite il codice hash della chiave |
| Queue | Lista FIFO di elementi |
| ReadOnlyCollectionBase | Classe base astratta per collezioni fortemente tipizzate, read-only e non generiche |
| SortedList | Rappresenta una sequenze di coppie chiave-valore ordinata ed accessibile sia tramite la chiave che tramite l'indice. |
| Stack | Classica struttura LIFO |
| StructuralComparison | Effettua una comparazione strutturale tra due collezioni di oggetti. |

Ecco le interfacce

| Interfaccia | Uso |
|------------------------------|---|
| ICollection | Definisce i metodi da utilizzare per manipolare collezioni generiche |
| IComparer | Definisce i metodi per la comparazione di due oggetti |
| IDictionary | Rappresenta una generica collezione aventi come elementi coppie chiave-valore |
| IDictionaryEnumerator | Enumera gli elementi di un Dictionary non generico |
| IEnumerator | Permette l'iterazione sugli elementi di una collezione. Molto importante. |
| IEnumerable | Restituisce l'interfaccia IEnumerator di una collezione |
| IEqualityComparer | Definisce dei metodi per la comparazione degli oggetti con uguaglianza |
| IList | Definisce una lista di oggetti accessibili tramite indice |

| | |
|--------------------------|---|
| IEqualityComparer | Fornisce metodi per la comparazione strutturale di collezioni |
| IEqualityComparer | Fornisce metodi che permettono la comparazione strutturale. |

La struttura invece è **DictionaryEntry** che definisce una coppia chiave valore da recuperare o inizializzare.

Le collezioni, tra le altre cose, vengono incontro alla necessità di superare alcuni punti deboli degli array come ad esempio l'incapacità di modificare le proprie dimensioni senza dover ricorrere ad artifici dispendiosi in termini di memoria. Si tratta di una struttura dati molto comoda e potente che vedremo meglio più avanti. Cosa quindi non va nell'esempio precedente? Sintatticamente nulla, il punto critico è la presenza o meglio la necessità di effettuare dei cast per operare un semplice assegnamento (senza di esso il compilatore si lamenta). Analogamente vi sono dei box-unbox impliciti nell'assegnare il valore alla variabile AL. La cosa funziona, per carità, ma il prezzo in termini prestazionali può essere significativo. Una delle osservazioni che gli sviluppatori fecero verso le versioni 1.0 ed 1.1 di .Net fu proprio questo fatto e la mancanza di qualche cosa di più duttile per gestire situazioni del genere. E, in generale, troppo spesso era necessario ricorrere a questa tecnica, il cast, potente e comoda ma, appunto, dispendiosa. I cast sono a volte necessari, non vanno certo demonizzati. Ma, quando possibile, è bene farne a meno aumentando il grado di informazioni che diamo al compilatore. I **generici** o **generics** nella letteratura inglese, rispondono a questa necessità, garantendo, tra l'altro, un miglior controllo in compile-time rispetto a quanto avviene con i cast, dove possiamo riscontrare incoerenze in particolare in execution-time (onestamente a me è capitato ben di rado ma è un problema teoricamente da non sottovalutare). Un ulteriore vantaggio si ha in termini prestazionali in quanto il compilatore se la cava meglio con i generici rispetto a quanto riesce a fare in situazioni di box-unbox dove tra l'altro entra in gioco anche il garbage collector, come noto. In pratica l'introduzione dei generics deve essere vista come un aspetto client oriented in molti casi; ovvero il creatore di una classe o di un metodo può permettere tramite essi al client di scegliere il tipo del parametro da passare senza ricorrere ad una generalizzazione che richiederebbe parametri di tipo object, a tutto vantaggio delle prestazioni e della sicurezza come è facile intuire.

La premessa che è doveroso fare riguarda la reale complessità dei generici che presentano parecchi "spigoli nascosti" e, nonostante possano apparire quasi banali per quanto riguarda il loro ruolo sono invece ricchi di sfumature tanto è che le specifiche fissate in .Net 2.0 sono molto dettagliate proprio per definire correttamente il loro funzionamento generale. In rete si trovano anche dei testi interi dedicati ai generici. Per fortuna, per il loro uso, molti dettagli possono essere tranquillamente ignorati.

I generici ricadono essenzialmente in due categorie: **tipi** e **metodi**. I primi sono sostanzialmente degli alias per i tipi noti (laddove tra questi ricadono anche le classi, le interfacce ed i delegati) e la notazione che viene adottata è costituita da una coppia <>. Per inciso sottolineiamo come in questo modo si realizzi un comportamento polimorfico per i tipi. Per entrare nell'argomento partendo dall'esempio precedente consideriamo che la classe System.Collections è composta da elementi fortemente tipizzati. Tuttavia il framework .Net prevede una controparte "generica" ovvero **System.Collections.Generic**. Questo namespace contiene gli equivalenti generici che accettano parametricamente i tipi proposti dall'utente senza necessità di cast od operazioni di boxing intermedie. L'esempio precedente diventa quindi:

| C# | Esempio 14.2 |
|----|-----------------------------------|
| 1 | using System.Collections.Generic; |
| 2 | |
| 3 | class Program |
| 4 | { |
| 5 | public static void Main() |
| 6 | { |
| 7 | List<int> AL = new List<int>(); |
| 8 | AL.Add(100); |
| 9 | int i = AL[0]; |
| 10 | } |
| 11 | } |

`List<T>` è definito in modo da accettare parametri senza ricorrere ad operazioni di box e così pure è possibile ricavare valori da esso, come alla riga 9, senza ricorrere ad un cast. Inoltre non è possibile inserire valori diversi da quelli di tipo intero a tutto garanzia di protezione da fastidiosi bugs. Un altro esempio simile è il seguente:

```
C# Esempio 14.3
1  using System;
2  using System.Collections.Generic;
3
4  class Program
5  {
6      public static void Main()
7      {
8          List < int > list = new List < int > ();
9          list.Add(100);
10         list.Add(200);
11         list.Add(300);
12         Mostra(list);
13         int x = list[1];
14         Console.WriteLine(x + 1);
15     }
16
17     public static void Mostra(List<int> list)
18     {
19         foreach (int i in list)
20             Console.WriteLine(i);
21     }
22 }
```

Pensate di dover applicare il `foreach` su di una struttura dati contenente qualche milione di elementi e pensate al risparmio in termini di operazioni box e unbox evitate... Questo è uno dei motivi per cui la controparte generica ha molto spesso preso il posto del namespace originario `System.Collections` che pure ancora oggi esiste non fosse altro che per compatibilità verso il passato.

Ma se utilizzare le classi del framework è molto comodo poter utilizzare queste funzionalità nelle nostre applicazioni lo è ancora di più. Come base prendiamo il seguente esempio:

```
C# Esempio 14.4
1  using System.Collections.Generic;
2  using System;
3
4  class Program
5  {
6      static void tipo <T>(T a)
7      {
8          Console.WriteLine(a.GetType());
9      }
10     public static void Main()
11     {
12         int x = 2;
13         tipo<int>(x);
14         string s1 = "ciao";
15         tipo<string>(s1);
16     }
17 }
```

Il metodo di nome "tipo" accetta parametricamente non solo i valori ma anche i tipi. Infatti la riga 13 passa un intero mentre la 15 una stringa. Una piccola nota di stile: l'uso della lettera T non è obbligatorio all'interno di <>, come è facile provare da soli, ma è convenzionalmente adottato e, per la chiarezza dei vostri programmi, consiglio di mantenere questa abitudine. Un sistema molto flessibile e comodo, come si può facilmente intuire. Con questo non

sono tutte rose e fiori, come avevamo anticipato ci sono delle zone d'ombra con cui ci si può scontrare all'improvviso... ad esempio:

```
C# Esempio 14.5
1  using System.Collections.Generic;
2  using System;
3
4  class Program
5  {
6      static void tipo <T>(T a, T b)
7      {
8          Console.WriteLine(a + b);
9      }
10     public static void Main()
11     {
12         int x = 2;
13         int y = 3;
14         tipo<int>(x, y);
15     }
16 }
```

Questo codice non compila “by design” e non è semplice trovare una soluzione (una possibilità è passare attraverso reflection, come vedremo ma non è particolarmente efficiente). Il compilatore non sa “a priori” quale sarà il tipo da sostituire a <T> per cui non è in grado di generare codice adatto a tutte le possibili situazioni. In questo senso i generics, saranno anche più belli e sicuri ma, a parer mio, perdono il confronto con i template del C++. Insomma usare <T> non è la panacea di tutti i mali ma va fatto quando serve e dopo aver esaminato a fondo il problema, almeno quando si parla di metodi. Parlando di generics e metodi viene utile segnalare un piccolo trucco che permette di rendere il codice più lineare. Nel caso in cui il metodo generico abbia dei parametri il compilatore è in grado di applicare l’inferenza sui parametri e quindi non è necessario specificare il tipo in maniera esplicita. Provate ad esempio a togliere <int> e <string> rispettivamente alla riga 13 ed alla 15 dell’esempio 14.4. Compila ugualmente. Sottolineo ancora che se il metodo generico non ha parametri è necessario specificare il tipo

Più interessante è vedere i generics applicati ad una **classe** o ad una **struct**. L’esempio classico in questi casi è quello della struttura che rappresenta un punto:

```
C# Esempio 14.6
1  using System;
2
3  public struct Punto<T>
4  {
5      private T x;
6      private T y;
7      private T z;
8
9      public Punto (T xx, T yy, T zz)
10     {
11         x = xx;
12         y = yy;
13         z = zz;
14     }
15 }
16
17 class Program
18 {
19     public static void Main()
20     {
21         Punto<int> p1 = new Punto<int>(3,4,5);
22         Punto<double> p2 = new Punto<double>(3.3, 3.4, 3.5);
23     }
24 }
```

L'esempio è abbastanza banale ma illustra sufficientemente il concetto. Tra l'altro basta cambiare la parola **struct** con **class** alla riga 3 per accorgersi che la cosa funziona perfettamente anche con le classi. Parametrizzazione completa, questa è l'idea che sta dietro ai generics e qui trova la sua perfetta applicazione. In questo caso i vantaggi non sono solo esprimibili in termini prestazionali ma in generale è possibile organizzare in modo più ampio le proprie applicazioni da un punto di vista architetturale. Ovviamente questo genere di costrutti viene naturale quando svolgiamo operazioni generalizzate quindi non legate ad un particolare tipo ed in scenari applicativi generalmente ampi e, appunto, generici. In alcuni casi ad avvantaggiarsi di questo modo di definire le classi è la riusabilità delle stesse. Lo schema costruttivo che estende un elemento nella sua controparte generica dovrebbe essere chiaro:

```
class x {} -> classe normale
class x <T> {} -> classe generica
```

Una classe generica è spesso usata come base per altre classi. In questo senso il C++ è più espressivo in termini di definizione laddove la parola template renda senza dubbio meglio l'idea. La regola principale che una classe derivata da una classe generica è che devono essere specificati i tipi parametri.

```
public class C<T>
{
}

public class D : C<int>
{
}
```

In questo caso la classe D deriva da C specificando il tipo, in questo caso un intero ma poteva essere ovviamente anche un altro. Un'altra regola da seguire è che se la classe generica presenta dei metodi virtuali o astratti la classe derivata deve effettuare l'override di tali metodi.

```
public class C<T>
{
public virtual void m(T x)
{
}
}

public class D : C<int>
{
public override m(int x)
{
}
}
```

nella sezione dedicata agli esempi vedremo come questo sia applicabile nella pratica.

Il concetto di genericità, chiamiamolo così, si applica ovviamente anche alle interfacce. Concettualmente non siamo molto lontani da quanto visto per le classi, come è abbastanza comprensibile considerando la vicinanza tra classi ed interfacce. La classe che deriva da una interfaccia generica deve implementare in toto l'interfaccia stessa:

```
public interface Itest<T>
{
T metodo1(T arg1);
T metodo2(T arg2);
}
```

che potremmo implementare come segue:

```
public class C: Itest<int>
{
public int metodo1(int arg1);
```

```
public int metodo2(int arg2);
}
```

Ancora, non possono mancare i **delegati** a questo elenco. In questo caso è possibile parametrizzare la chiamata ad un metodo.

```
public delegate void Del<T>(T arg)
.....
Del<int> del = new Del(int);
.....
Del<string> del2 = new Del<string>
```

Anche in questo caso deve essere eseguita l'implementazione, ormai il concetto dovrebbe essere chiaro. Il vantaggio è quello di avere una sorta di modello di delegato valido per chiamate a metodi con parametrizzazioni differenti, come nel piccolo frammento di codice presentato.

Interessante è anche l'uso che si fa, parlando dei generics, della keyword **where**. Il suo scopo, come definito è quello di stabilire dei vincoli che sui tipi che si vogliono utilizzare come argomenti per un parametro di tipo definito in una dichiarazione generica. Il motivo per cui è a volte necessario ricorrere a queste limitazioni è dato dal fatto che un parametro unbound (così si definisce in assenza di vincolo) deve avere membri che discendono direttamente da System.Object. In presenza di metodi personalizzati si possono presentare errori in fase di compilazione qualora vi siano richiami a tali metodi in presenza di generics non ancora noti al compilatore. I vincoli possibili, che servono a risolvere la problematica descritta, sono riassunti nella seguente tabella:

| | |
|--------------------------|---|
| where T: struct | Il tipo parametro <T> deve essere tipo System.ValueType |
| where T: class | Il tipo parametro <T> è reference type |
| where T: new() | Il tipo parametro <T> deve avere un costruttore di default |
| where T:nome-classe-base | Il tipo parametro <T> deve discendere dalla classe base |
| where T:nome interfaccia | Il tipo parametro <T> deve implementare l'interfaccia specificata |

Possiamo osservare il seguente esempio che, facendo uso della libreria matematica Math già citata più indietro, calcola l'elevamento a potenza di un numero per un altro:

```
C# Esempio 14.7
1 using System;
2 using System.Globalization;
3
4 class DueNumeri<T> where T:IConvertible
5 {
6     public T X;
7     public T Y;
8
9     public DueNumeri(T x, T y)
10    {
11        X = x;
12        Y = y;
13    }
14    NumberFormatInfo fmt = NumberFormatInfo.CurrentInfo;
15
16    public double Pot(DueNumeri<T> dn)
17    {
18        return (Math.Pow(X.ToDouble(fmt), Y.ToDouble(fmt)));
19    }
20 }
21
22 class Program
23 {
24     public static void Main()
25     {
26         DueNumeri <double>dn = new DueNumeri<double>(3.0, 4.0);
```

```

27         Console.WriteLine(dn.Pot(dn));
28
29     }
30 }

```

Il punto cruciale è ovviamente costituito dalla parte della riga 4 evidenziata in rosso che corrisponde al quinto punto della tabella precedente. Provate a togliere la parte in rosso e ricompilate. Vi beccherete una rispostaccia da parte del compilatore. Perché? Il problema nasce quando viene preso in esame il metodo Pot all'interno del quale vi è la richiesta di operare un elevamento a potenza (il metodo Pow è contenuto all'interno di Math come ricorderete e vuole due double come argomenti) utilizzando come argomenti X e Y che, al momento della compilazione, non sono noti. Ovvero il compilatore non può sapere se avrà a che fare con interi, double, stringhe o che altro per cui, nel dubbio, richiede ulteriori informazioni. La cosa, è evidente, non si può nemmeno cercare di risolvere attraverso un cast sempre per lo stesso motivo cioè che non sappiamo su che tipo di dato opereremo il nostro cast ed anche qui a compile time avremo un chiaro messaggio di errore. Per risolvere questa situazione ci preoccupiamo di tranquillizzare il compilatore informandolo che <T> comunque dovrà implementare l'interfaccia IConvertible che tra i suoi metodi ha proprio ToDouble. Questo procedimento è molto utile in svariate situazione anche se, di primo acchito, toglie un po' quel senso "generalista" che si ha quando creiamo un tipo generico.

METODI ANONIMI

In realtà con l'argomento precedente i **metodi anonimi** c'entrano poco in linea teorica ma mi piace riunirli insieme perchè si tratta di una seconda più novità, introdotta con .Net 2.0 quando il framework e con esso C# cominciarono davvero ad assumere l'aspetto per così dire adulto. In realtà con C# 3.0 verranno introdotte le lambda-expressions che subentreranno ai metodi anonimi. Le linee guida suggeriscono di utilizzare proprio le lambda expressions invece dei metodi anonimi in ragione della sintassi più leggera e della miglior manutenibilità che esse offrono. Tuttavia anche i metodi anonimi sono pienamente supportato ed utilizzabili per cui meritano di essere ricordati.

Anche in questo caso, se vogliamo trovare un filo comune, si tratta in qualche modo di una sorta di "generalizzazione", i puristi di C# perdonino questa semplificazione intellettuale. In pratica si tratta di questo: quando definiamo un delegato, abbiamo visto, dobbiamo anche passargli un metodo che condivida la medesima signature specificata nel delegato. Attraverso un metodo anonimo siamo in grado di usare un blocco di codice, appunto anonimizzato, come parametro ad un delegato.

Il seguente esempio è una versione ridotta del 12.1:

```

C#   Esempio 14.8
1   using System;
2   public delegate int OperaSuInteri (int x, int y);
3
4   public class Test
5   {
6       public static void Main()
7       {
8           OperaSuInteri Osi = delegate(int x, int y)
9           {
10              return(x + y);
11          };
12          Console.WriteLine(Osi(2, 3));
13      }
14 }

```

Cosa cambia in sostanza? Nulla in fase di definizione del delegato. Al momento della sua definizione tuttavia invece di passargli un metodo forniamo un blocco di codice premettendo la parola delegate (riga 8) e ovviamente conservando parametrizzazione ed il valore di ritorno. Da notare il punto e virgola con cui termina il blocco che in pratica inizia con la parola delegate stessa. Alcune semplici regole vanno rispettate: non è possibile definire salti che escano da un

blocco di un metodo anonimo ed analogamente non è possibile saltarvi dentro. Non è possibile usare codice unsafe all'interno di un metodo anonimo. Parametri ref ed out definiti esternamente al metodo anonimo sono pure inaccessibili.

I metodi anonimi forniscono un certo vantaggio in termini di stesura del codice. Ma, come detto, al loro posto è meglio usare le espressioni oggetto del prossimo sottoparagrafo che presentano vantaggi implementativi ed espressivi, in termini di leggibilità, abbastanza marcati e che evidenzieremo qui di seguito

LAMBDA EXPRESSIONS

Introdotte con C# 3.0 si tratta ancora di funzioni anonime che posso essere usate per creare delegati o “expression trees”, di questi ultimi parleremo altrove. L'operatore che viene usato per introdurre le lambda expressions è => (simbolo il cui nome è appunto “lambda”) che possiamo rendere come “fino a”. Formalmente abbiamo:

(parametri) => espressione o sequenza di istruzioni.

La parentesi è opzionale nel caso ci sia un solo parametro. Partiamo subito con un po' di codice poi ne discuteremo. L'esempio è in pratica lo stesso che si trova su **MSDN** e lo uso in quanto non è facile trovarne uno più basilare:

```
C# Esempio 14.9
1 using System;
2
3 public class Test
4 {
5     delegate int del(int i);
6     public static void Main()
7     {
8         del myDelegate = x => x * x;
9         int y = myDelegate(5);
10        Console.WriteLine(y);
11    }
12 }
```

La riga 8 è una tipica espressione, non replicabile se usiamo le funzioni anonime che prevedono sempre e comunque un blocco di codice. A questo proposito vediamo invece un esempio con più di una istruzione:

```
C# Esempio 14.10
1 using System;
2
3 public class Test
4 {
5     delegate int del(int i);
6     public static void Main()
7     {
8         del myDelegate = x =>
9         {
10            x = x + 1;
11            return x * x;
12        };
13        int j = myDelegate(5);
14        Console.WriteLine(j);
15    }
16 }
```

E il risultato stavolta sarà 36 e non più 25 come nel 14.9. Aggiungiamo ora un ulteriore esempio con più di un parametro:

```

C# Esempio 14.11
1 using System;
2
3 public class Test
4 {
5     delegate int del(int i, int j);
6     public static void Main()
7     {
8         del myDelegate = (x, y) =>
9         {
10            x = x + y;
11            return x * x;
12        };
13        int j = myDelegate(5,3); //j = 25
14        Console.WriteLine(j);
15    }
16 }

```

Stavolta il risultato sarà 64. È evidente che i parametri devono corrispondere ai parametri del delegato mentre il valore di ritorno è dello stesso tipo di quello che determina il delegato. Risulta immediato notare come i parametri siano gestiti in maniera inferenziale dal compilatore. Infatti alla riga 8 abbiamo scritto (x, y) e non (int x, int y), appunto perchè è possibile utilizzare su di essi **l'inferenza di tipo**. In caso di possibili ambiguità invece dobbiamo specificare il tipo stesso, come nel seguente frammento di codice che si riferisce alla riga 8 dell'esempio 14.10:

```

del myDelegate = (int x) =>
.....

```

Questa possibilità di usare l'inferenza è un'altra carenza delle funzioni anonime rispetto alle espressioni lambda. Interessante è il comportamento delle espressioni lambda di fronte a valori esterni.

```

C# Esempio 14.11
1 using System;
2
3 public class Test
4 {
5     delegate int Function (int i);
6
7     public static void Main()
8     {
9         int x = 3;
10        Function multix = n => n * x;
11        x = 5;
12        Console.WriteLine(multix(5));
13    }
14 }

```

In questo caso la variabile n viene detta catturata mentre una espressione lambda che cattura una variabile viene detta chiusura. Come è evidente dall'esempio la variabile viene valutata quando il delegato viene invocato non quando la variabile viene catturata. Il risultato esposto a video infatti è 25, non 15.

Scendiamo un po' in dettaglio. Abbiamo parlato di "chiusura". In ambito informatico una chiusura è una funzione di prima classe dotata di variabili libere ma limitate in un certo ambito. Una funzione viene detta di prima classe in sostanza se può essere usata come argomento o valore di ritorno per altre funzioni, se può essere memorizzata in una collezione o se è possibile creare da esse nuove funzioni a runtime o assegnarle ad una variabile. Il C classico ad esempio non può creare nuove funzioni dinamicamente quindi non ha funzioni di prima classe. Il concetto di chiusura è dato proprio dalle limitazioni alla quale sono soggette le variabili libere, limitazioni determinate quindi da un ambito operativo. Un esempio molto semplice è il seguente

C# Esempio 14.12

```
1 using System;
2
3 public class Test
4 {
5     delegate int Function (int i);
6
7     public static void Main()
8     {
9         int y = 2;
10        Function multix = x =>
11        {
12            y = y + 1;
13            return x * y;
14        };
15        var a = multix(5);
16        var b = multix(6);
17        Console.WriteLine(a);
18        Console.WriteLine(b);
19    }
20 }
```

L'output è il seguente:

15

24

E non 15 e 18 come forse qualcuno si poteva aspettare. In effetti il comportamento è un po' strano in quanto le variabili di tipo value, quali ad esempio gli interi, sono create, poste nello stack e poi eliminate, per cui ci si poteva aspettare che se la prima moltiplicazione era $5 * 3$ la seconda fosse $6 * 3$ e non $6 * 4$ come invece è. Questa situazione è determinata dalla chiusura. Internamente la cosa viene gestita, detto un po' semplicisticamente, da parte del compilatore tramite la creazione di una classe che contiene anche la variabile sulla quale viene applicato il metodo, in questo caso il metodo multix, collegato al delegato.