Generics

Introduzione ai Generics

Progetto di Informatica classe 4^a

Ambiente: .NET 4.0/C# 4.0

Anno 2016

Indice generale

1 Introduzione

Questo tutorial fornisce un'introduzione sul funzionamento dei *generics:* la funzionalità del linguaggio che consente di implementare strutture dati e algoritmi generici.

1.1 Cosa si intende per “generico”

Innanzitutto è importante comprendere il significato del termine. Il vocabolario fornisce alcune definizioni, che riporto parzialmente:

> *“… non specifico, non specificato... non particolare... privo di particolari proprietà… che manca di determinatezza, di concretezza…”*

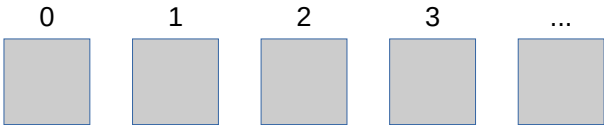
Il termine fa dunque riferimento alla vaghezza, alla mancanza di specificità e di dettagli, etc.

In alcuni casi ciò assume connotati negativi: “sei troppo generico… cerca di essere più preciso!”. Ma vi sono situazioni nelle quali esprimersi in termini generici risulta utile, poiché consente di esprimere idee, concetti e operazioni senza perdersi in dettagli irrilevanti.

1.2 Strutture dati generiche

Quando si parla di strutture dati generiche si intende *dare importanza all'organizzazione dei dati, indipendentemente dalla loro natura* (numeri, stringhe, date, etc).

Ad esempio, la figura schematizza una lista di oggetti; ogni elemento della lista ha una posizione rispetto agli altri: il primo, il secondo, il terzo, etc.



È l'organizzazione degli elementi a caratterizzare la lista, non certo il loro tipo. Una lista di interi ha l'identica struttura di una lista di stringhe, o altro; la natura del singolo elemento è irrilevante.

1.3 Algoritmi generici

Alcuni procedimenti non dipendono dal tipo dei dati, ma soltanto dalle operazioni eseguite su di essi. Ad esempio:

- ***Ricerca***: si scorre la lista degli elementi fino a trovare quello ricercato.
- ***Ordinamento***: si confronta ogni elemento con gli altri, scambiandoli di posto se non sono nell'ordine richiesto.
- **Filtro**: si scorre la lista degli elementi e si crea una nuova lista contenente soltanto quelli che rispettano una determinata condizione.

In queste definizioni ho usato il termine *elemento*; non ho parlato di numeri, stringhe, date, etc, poiché il risultato dell'algoritmo dipende soltanto dalle operazioni eseguite.

2 Implementare strutture dati generiche

Il modo migliore per comprendere i *generics* è quello di implementarli; prima, però, è opportuno "toccare con mano" il tipo di scenario nel quale sono utili.

2.1 Liste in un mondo senza *generics*

Supponiamo che C# non abbia i *generics* e pertanto non abbia nemmeno il tipo **List<>**[1]. Per gestire liste di dati abbiamo a disposizione soltanto i vettori, i quali hanno però una dimensione prefissata. Se vogliamo utilizzare una lista dinamica doppiamo implementarla.

2.1.1 Lista dinamica di interi

Vogliamo realizzare una lista dinamica di interi; dovrà fornire le seguenti funzionalità:

- Inserimento in coda di un valore.
- Accesso a un valore dato il suo indice.
- Accesso al numero dei valori memorizzati.

Ecco il codice:

```
public class IntList
{
    private int[] items = new int[10];
    public int Count { get; private set; }

    public void Add(int item)
    {
        if (Count == items.Length) // vettore pieno -> raddoppia la sua dimensione
            Array.Resize(ref items, items.Length*2);
        items[Count] = item;
        Count++;
    }
    public int GetItem(int index)
    {
        return items[index];
    }
}
```

Segue un frammento di codice che usa la lista:

```
IntList list = new IntList();
list.Add(10);
list.Add(20);
list.Add(30);
for (int i = 0; i < list.Count; i++)
{
    Console.WriteLine(list.GetItem(i));  // -> 10, 20, 30
}
```

1 Era così nella versione C# 1.0. Esisteva comunque un'alternativa all'uso dei *generics*, ma meno efficiente e semplice da utilizzare.

`IntList` funziona, ma ci permette di gestire soltanto interi.

2.1.2 Lista dinamica di stringhe

Deve fornire le stesse funzionalità di `IntList`.

```
public class StringList
{
    private string[] items = new string[10];
    public int Count { get; private set; }

    public void Add(string item)
    {
        if (Count == items.Length) // vettore pieno -> raddoppia la sua dimensione
            Array.Resize(ref items, items.Length*2);
        items[Count] = item;
        Count++;
    }
    public string GetItem(int index)
    {
        return items[index];
    }
}
```

Ecco le due classi a confronto (le differenze sono state evidenziate):

```
public class IntList
{
    private int[] items = ...
    public int Count { get; private set; }
    public void Add(int item) {}
    public int GetItem(int index) {}
}
```

```
public class StringList
{
    private string[] items = ...
    public int Count { get; private set; }
    public void Add(string item) {}
    public string GetItem(int index) {}
}
```

L'idea che emerge confrontando `IntList` e `StringList` è chiara: se desideriamo collezioni di dati appartenenti a vari tipi, dobbiamo scrivere una lista per ogni tipo, nonostante le classi realizzate siano praticamente uguali tra loro.

2.2 Una lista generica

Cos'è che differenzia `IntList` da `StringList`? Solo il tipo degli elementi, poiché le operazioni compiute su di essi sono identiche. Ebbene, C# consente di implementare una lista senza specificare il tipo degli elementi:

```
public class GenericList<T>
{
    private T[] items = new T[10];
    public int Count { get; private set; }

    public void Add(T item)
    {
        if (Count == items.Length)
```

```
            Array.Resize(ref items, items.Length*2);
        items[Count] = item;
        Count++;
    }
    public T GetItem(int index)
    {
        return items[index];
    }
}
```

La classe ha l'identica struttura di `IntList` e `StringList`, ma al posto del tipo degli elementi usa un segnaposto chiamato ***parametro di tipo***. (Per convenzione si usa il nome **T**.)

L'intestazione della classe è fondamentale, poiché è ciò che la rende generica:

```
public class GenericList<T>   // T fa da sostituto per il tipo degli elementi
{
    ...
}
```

L'uso di un *parametro di tipo* stabilisce che il tipo degli elementi non è specificato e dunque può essere uno qualsiasi. Una classe simile si definisce ***open type***.

2.2.1 Uso della lista generica: closed types

Un *tipo aperto* non può essere utilizzato per dichiarare variabili. Infatti, una variabile deve avere un tipo specifico. Per utilizzare la lista occorre dichiarare il tipo degli elementi:

```
var list = new GenericList<string>();      // argomento di tipo: string
list.Add("qui");
list.Add("quo");
list.Add("qua");

for (int i = 0; i < list.Count; i++)
{
    Console.WriteLine(list.GetItem(i));
}

var listI = new GenericList<int>();        // argomento di tipo: int
listI.Add(10);
listI.Add(20);
listI.Add(30);
for (int i = 0; i < listI.Count; i++)
{
    Console.WriteLine(listI.GetItem(i));
}
```

`GenericList<string>` e `GenericList<int>` sono chiamati ***closed types***, e sono tipi di dati a tutti gli effetti. Il tipo specificato tra parentesi angolari si chiama ***argomento di tipo***.

2.3 Tipi generici con più parametri di tipo

I tipi generici possono specificare più *parametri di tipo*. Un esempio è **Dictionary<,>**: una collezione di coppie chiave-valore. Segue una versione semplificata dell'intestazione:

```
public class Dictionary<TKey,TValue>
{
    ...
}
```

L'uso di nomi significativi per i parametri di tipo, **TKey** e **TValue**, suggerisce la loro funzione. È una convenzione molto utile, usata in tipi e metodi generici con più *parametri di tipo*.

"Specializzazione" dei *generics*

Per ogni uso della classe generica, il compilatore **JIT** genera una classe concreta sostituendo al *parametro di tipo* il tipo effettivo. Questa operazione viene definita: *specializzazione del generic*.

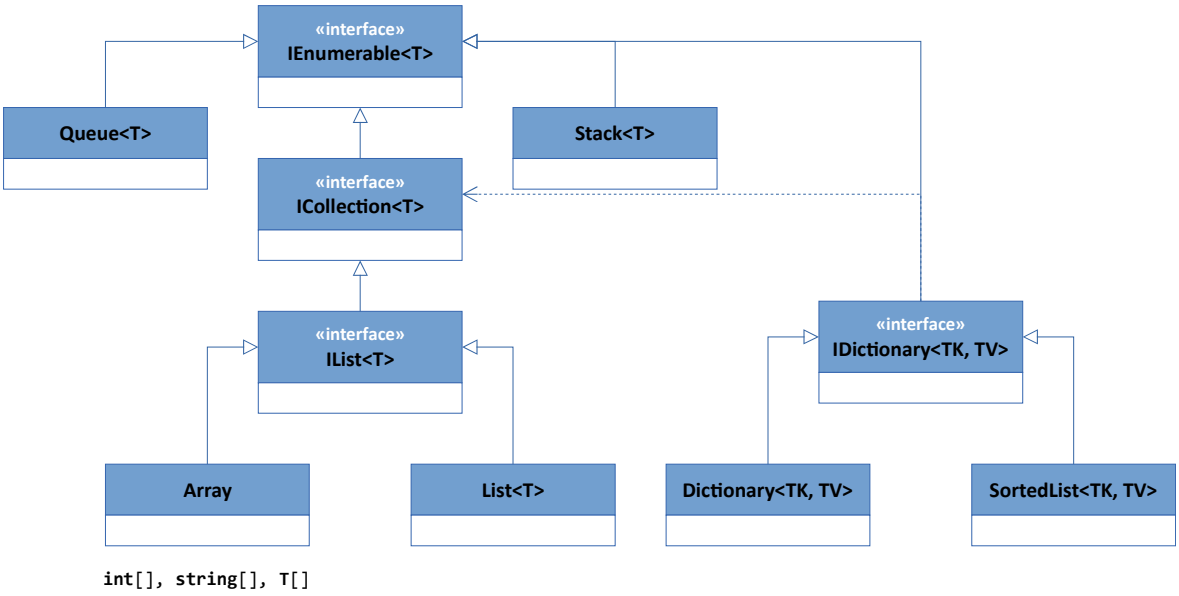
Dunque, un programma che esegue le istruzioni:

```
GenericList<string> listString = new GenericList<string>();
GenericList<int> listInt = new GenericList<int>();
```

contiene in realtà tre classi. La classe generica e due classi normali generate automaticamente dal JIT, una che usa il tipo **string**, l'altro il tipo **int**.

3 Tipi generici predefiniti:

Esistono molti tipi generici predefiniti, sia concreti (classi e strutture), che astratti (interfacce e classi astratte). Qui mi limito a mostrarne alcuni[2] che implementano collezioni generiche.



Brevemente:

- **IEnumerable<>**: sequenza di elementi ai quali si può accedere dal primo all'ultimo.
- **ICollection<>**: collezione dinamica di elementi. Non è indicizzabile.
- **IList<>**: collezione dinamica di elementi indicizzabile.
- **List<>**: implementa una collezione dinamica indicizzabile.
- **Queue<>**: implementa una *coda*.
- **Stack<>**: implementa una *pila*.
- **IDictionary<,>**: dizionario, e cioè una collezione di coppie chiave→ valore.
- **Dictionary<,>**: implementa un dizionario. I valori sono accessibili attraverso le chiavi corrispondenti.
- **SortedList<,>**: implementa un dizionario ordinato in base alle chiavi. Fornisce anche l'accesso indicizzato.

2 Tra gli altri, ho omesso i tipi **ICollectionReadOnly<>** e **IListReadOnly<>**, **ObservableCollection<>** e **Collection<>**.

4 Metodi generici

Un metodo generico implementa un algoritmo che non dipende dal tipo dei dati processati.

4.1 Ottenere una stringa da un vettore di elementi

Supponiamo di volere realizzare un procedimento in grado di produrre i seguenti risultati:

```
int[] interi = {1, 2, 3, 4, 5};
string[] str = { "qui", "quo", "qua" };
double[] reali = { 1.3, 23.24, 10 };
```

```
"1,2,3,4,5"
"qui | quo | qua"
"1.3; 23.24; 10"
```

Il procedimento produce una stringa con gli elementi di un vettore, utilizzando un separatore a scelta. Si tratta di un algoritmo generico, poiché il risultato non dipende dal tipo degli elementi[3].

4.1.1 Versione tipizzata: uso del tipo int

Innanzitutto implemento una versione che utilizza un vettore di interi:

```
public class GenericMethods
{
    public static string ToString(int[] items, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (var item in items)
        {
            if (sb.Length > 0)    // evita l'inserimento del separatore all'inizio
                sb.Append(separator);
            sb.Append(item);
        }
        return sb.ToString();
    }
}
```

Il metodo usa uno **StringBuilder** per costruire la stringa. Il metodo **Append()** invoca **ToString()** su `item` per ottenere la stringa corrispondente. Poiché qualunque tipo di dato definisce **ToString()**, ne segue che l'intero metodo è indipendente dal tipo del vettore. Possiamo dunque implementarlo come metodo generico.

4.1.2 Implementazione generica del procedimento

La versione generica del metodo contiene esattamente lo stesso codice:

```
public static string ToString<T>(T[] items, string separator)
{
    ...
}
```

L'unica novità riguarda l'intestazione; infatti, ciò che rende generico il metodo è il suo nome, comprendente il *parametro di tipo* **T**.

3 Purché il tipo dell'elemento sia convertibile in stringa. Ciò è sempre vero, dato che ogni tipo deriva da **object**, il quale definisce il metodo **ToString()**.

4.1.3 Uso dei metodi generici

La chiamata di un metodo generico specifica l'*argomento di tipo*, indicato tra parentesi angolari:

```
int[] interi = { 1, 2, 3 };
double[] reali = { 10.34, 12.1, 100 };

string si = GenericMethods.ToString<int>(interi, ",");       //-> "1,2,3"
string sd = GenericMethods.ToString<double>(reali, "| ");    //-> "10,34| 12,1| 100"
```

Vale però una regola: l'*argomento di tipo* deve essere compatibile con il *parametro di tipo* corrispondente. Dunque, la seguente istruzione è scorretta:

```
int[] interi = { 1, 2, 3 };
double[] reali = { 10.34, 12.1, 100 };

string si = GenericMethods.ToString<int>(reali, ","); // parametro:int   argomento:double
```

Infatti, l'*argomento di tipo* è **int**, mentre al metodo viene passato un vettore **double**.

4.1.4 Inferenza dell'argomento di tipo

Nella maggior parte dei casi non serve specificare l'*argomento di tipo*, poiché il linguaggio può dedurlo dal tipo del parametro.

Nel seguente codice viene eseguita la chiamata a `ToString<int>()`, poiché il vettore è di tipo **int**:

```
int[] interi = { 1, 2, 3 };
string si = GenericMethods.ToString(interi, ",");    // C# deduce ToString<int>()
```

Ciò rende possibile utilizzare i metodi generici con la consueta sintassi.

4.2 Metodi generici statici e d'istanza

Un metodo generico può essere sia statico che d'istanza. Se è un metodo di istanza e appartiene a un tipo generico, il nome del *parametro di tipo* del metodo non può coincidere con quello del *parametro di tipo* della classe.

5 Vincoli sui parametri di tipo

Nei due esempi forniti, tipo `GenericList<>` e metodo `ToString<>`, è possibile applicare i *generics* a qualsiasi tipo di dato. Infatti, le uniche operazioni eseguite sulle variabili di tipo generico sono l'assegnazione e **ToString()**: poiché qualunque tipo di dato ammette tali operazioni, qualunque tipo di dato può essere usato in entrambi i casi.

Ma ci sono operazioni che possono essere applicate soltanto a determinati tipi di dati; in questo caso è necessario imporre dei vincoli sui *parametri di tipo*, in modo da restringere l'applicabilità del *generic*.

5.1 Metodo di ordinamento generico: “tipi comparabili”

Consideriamo l'ipotesi di implementare un metodo di ordinamento generico:

```
void Sort<T>(T[] items)
{
    for (int i = 0; i < items.Length - 1; i++)
    {
        for (int j = i+1; j < items.Length; j++)
        {
            if (items[i] > items[j])  // ">" non è utilizzabile con qualsiasi tipo!
            {
                T tmp = items[i];
                items[i] = items[j];
                items[j] = tmp;
            }
        }
    }
}
```

In sostanza, l'ordinamento non è un procedimento completamente generico: non tutti i tipi di dati hanno gli operatori di confronto. In generale: *non tutti i tipi implementano la comparazione tra due valori*.

Se vogliamo usare il metodo con quei tipi che la implementano (tutti i tipi primitivi, ad esempio), è necessario imporre tale vincolo al parametro di tipo:

```
void Sort<T>(T[] items) where T:IComparable<T>   // T deve esse "comparabile"!
{
    for (int i = 0; i < items.Length - 1; i++)
    {
        for (int j = i+1; j < items.Length; j++)
        {
            if (items[i].CompareTo(items[j]) > 0)  // usa il metodo: A.CompareTo(B)
            {
                ...
            }
        }
    }
}
```

L'intestazione del metodo contiene la clausola **where**; questa rappresenta un vincolo e stabilisce che **T** *deve implementare* l'interfaccia **IComparable<>**.

Posto questo vincolo, è possibile considerare gli elementi di tipo **T** come appartenenti a un tipo comparabile. Ciò consente di usare **CompareTo()**, definito da tutti i tipi comparabili:

```
if (items[i].CompareTo(items[j]) > 0) // A.CompareTo(B)
{
    ...
}
```

5.1.1 Elenco dei vincoli: clausola where

Segue l'elenco completo dei vincoli che è possibile utilizzare.

Caratteristica	Descrizione
where T: struct	L'*argomento di tipo* deve essere di *tipo valore* (struttura).
where T: class	L'*argomento di tipo* deve essere un *tipo riferimento* (classe).
where T: new()	L'*argomento di tipo* deve definire il costruttore senza parametri.
where T: <tipo>	L'*argomento di tipo* deve essere del tipo specificato o derivare da esso. (O implementarlo se <*tipo*> è un'interfaccia)
where T: U	L'*argomento di tipo* per T deve corrispondere all'argomento di tipo per U, o derivare da esso.

5.2 *Vincoli di tipo* all'opera

Il seguente codice mostra che il vincolo restringe l'uso del metodo ai soli tipi comparabili:

```
class Rider
{
    public string FullName { get; set; }
}
...
int[] values = { 4, 1, 6, 5 };
GenericMethods.Sort(values); //OK -> 1, 4, 5, 6
...
Rider[] riders = {
    new Rider { FullName= "Valentino, Rossi"},
    new Rider { FullName= "Jorge, Lorenzo"},
    new Rider { FullName= "Marc, Marquez"}
};

GenericMethods.Sort(riders); // errore: Rider non è un tipo comparabile!
```

5.3 Conclusioni

L'uso di vincoli appare un limite dei *generics*, ma è proprio questa funzionalità che aumenta i tipi e gli algoritmi generici che è possibile realizzare. Infatti, le sole operazioni eseguibili su qualsiasi tipo di dato sono l'assegnazione e **ToString()**[4]; limitandoci a queste, sono ben pochi i *generics* che potremmo implementare.

D'altra parte l'uso di vincoli impone dei costi. Ad esempio, se vogliamo utilizzare il metodo `Sort<>` su un vettore di `Rider`, è necessario rendere `Rider` un tipo comparabile, e cioè implementare l'interfaccia **IComparable<>**.

Alternativamente è possibile utilizzare un *delegate* generico.

4 In realtà a **ToString()** si aggiungono gli altri metodi del tipo **object**.

6 Delegate generici

I *delegate* sono tipi di oggetti che referenziano metodi. (Vedi tutorial **Delegate**.) Anche i *delegate* possono essere generici e cioè rappresentare metodi che definiscono parametri di tipo qualsiasi.

Non esiste alcun bisogno di definire nuovi *delegate*, poiché quelli predefiniti coprono qualsiasi necessità; qui intendo mostrare la loro utilità nella definizione di metodi generici.

6.1 Ordinare anche i tipi non comparabili: Comparison<>

Gli algoritmi di ordinamento si basano su un requisito fondamentale: disporre di un criterio che stabilisca il maggiore tra due elementi, e cioè la "comparabilità". È possibile implementare questa caratteristica nel tipo di dato che si desidera ordinare, oppure fornire al metodo di ordinamento il codice da usare per confrontare due elementi.

Il *delegate* generico **Comparison<>** rende possibile la seconda alternativa:

```
public delegate int Comparison<T>(T x, T y);
```

Il delegate è compatibile con qualsiasi metodo che definisce due parametri dello stesso tipo e restituisce un valore intero.

Ecco come utilizzarlo per rendere **Sort<>** applicabile a qualsiasi tipo di dato:

```
public static void Sort<T>(T[] items, Comparison<T> comparer)
{
    for (int i = 0; i < items.Length - 1; i++)
    {
        for (int j = i+1; j < items.Length; j++)
        {
            if (comparer(items[i], items[j]) > 0)
            {
                T tmp = items[i];
                items[i] = items[j];
                items[j] = tmp;
            }
        }
    }
}
```

6.1.1 Uso del un metodo con delegate generico

Questa versione di **Sort<>** è completamente generica e priva di vincoli[5]; infatti la responsabilità di confrontare due elementi è adesso del codice chiamante. Questo deve specificare il metodo di confronto, ad esempio mediante una *lambda expression*:

```
Rider[] riders = {
    new Rider { FullName= "Valentino, Rossi"},
    new Rider { FullName= "Jorge, Lorenzo"},
    new Rider { FullName= "Marc, Marquez"}
```

5 Non completamente; infatti il metodo si aspetta un vettore, e dunque non è in grado di elaborare liste e altre collezioni indicizzabili. Si può risolvere il problema dichiarando il parametro: **IList<T>**.

```
};

GenericMethods.Sort(riders, (r1, r2) => r1.FullName.CompareTo(r2.FullName));
```

Alternativamente, è possibile passare come argomento un metodo che confronta due `Rider`:

```
Rider[] riders = {
    new Rider { FullName= "Valentino, Rossi"},
    new Rider { FullName= "Jorge, Lorenzo"},
    new Rider { FullName= "Marc, Marquez"}
};

GenericMethods.Sort(riders, CompareRider);

...
static int CompareRider(Rider r1, Rider r2)
{
    return r1.FullName.CompareTo(r2.FullName);
}
```

Tipi, metodi e *delegate* generici sono il fondamento di LINQ, un linguaggio di interrogazione e manipolazione dei dati che consente di processare sequenze di elementi di qualsiasi tipo.

(Vedi tutorial **Linq**.)