Iteratori e sequenze

Progetto di Informatica classe 4^{a}

Ambiente: .NET 4.0/C# 4.0

Anno 2016

Indice generale

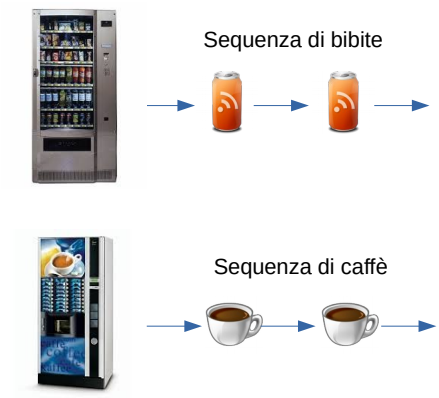
1 Introduzione

Gli i**teratori** sono molto importanti nella programmazione, anche se spesso li utilizziamo senza esserne consapevoli, ogni qual volta scriviamo un `foreach`, ad esempio. Questo tutorial ha lo scopo di svelarne il funzionamento e di mostrarne la centralità nel sistema di classi di .NET.

1.1 *Sequenze di oggetti*

È innanzitutto importante comprendere il concetto di *sequenza di oggetti*, distinto da quello di *collezione di oggetti*. Consideriamo due macchine molto simili: un distributore di bibite e uno di caffè. Entrambe distribuiscono bevande e sono utilizzate in modo pressoché identico, ma sono realizzate in modo diverso. Il primo gestisce una *collezione* di bibite, mentre il distributore di caffè *non contiene affatto una collezione di caffè*! Questi, infatti, sono prodotti man mano che vengono richiesti.

Ricapitolando: una *collezione* è un insieme di oggetti dotato di una determinata organizzazione. L'accesso ad essi, uno per volta, produce una *sequenza* di oggetti. È ciò che fa la macchina delle bibite. Ma anche la macchina del caffè produce una sequenza di oggetti, nonostante non ci sia nessuna collezione di caffè dentro la macchina.



Il confronto tra le due macchine mostra che *sequenza* è un concetto più generale di *collezione*. Ed anche è il più importante: in molti scenari si richiede di elaborare una sequenza di oggetti *indipendentemente dal fatto che siano memorizzati in una collezione oppure no*.

1.2 Iteratore

Un *iteratore* è un oggetto in grado di produrre una sequenza di oggetti. Gli iteratori hanno una caratteristica fondamentale: il loro uso *non dipende dall'origine degli oggetti che producono*. Di seguito vedremo come .NET consente di implementarli.

2 IEnumerator<T> e IEnumerable<T>

In .NET gli iteratori sono implementati attraverso l'interfaccia: `IEnumerator<T>`, dove `T` è il tipo degli oggetti prodotti. Esistono due modi per realizzarli, il primo dei quali è ormai obsoleto; ma è importante analizzarlo per comprendere la natura e il funzionamento di questi oggetti.

2.1 Implementare IEnumerator<T>

`IEnumerator<>` è un'interfaccia generica che stabilisce le caratteristiche di un iteratore (soltanto la proprietà `Current` e il metodo `MoveNext()` sono importanti):

```
public interface IEnumerator<T>: IEnumerator, IDisposable
{
    T Current { get; }
    bool MoveNext();
    object Current { get; } // ereditata da IEnumerator (obsoleta)
    void Reset();           // non è obbligatorio implementarlo
    void Dispose();         // definito da IDisposable (non è obbligatorio implementarlo)
}
```

L'uso di un iteratore è semplice: si invoca `MoveNext()` finché non restituisce `false`, mentre si usa `Current` per accedere a ogni elemento della sequenza:

```
var it = <crea iteratore>;
while (it.MoveNext() == true)  // -> si sposta al successivo elemento, se esiste
{
    <usa> it.Current;  // -> restituisce un elemento della sequenza
}
```

2.1.1 Esempio di un iteratore

Di seguito mostro un iteratore che restituisce una sequenza di interi casuali. Il costruttore richiede il numero dei valori da generare e l'intervallo all'interno del quale far cadere ogni valore[1]:

```
public class GeneratoreRnd : IEnumerator<int>
{
    private Random rnd = new Random();
    private int lunghezza, da, a;
    private int contaSequenza = -1;
    public GeneratoreRnd(int lunghezza, int da, int a)
    {
        this.da = da;
        this.a = a;
        this.lunghezza = lunghezza;
    }
    public int Current
    {
        get { return rnd.Next(da, a+1); }  // genera un valore nell'intervallo [da, a]
    }
```

1 Metodi e proprietà in grigio non sono necessari, ma devono essere definiti per rispettare la struttura dell'interfaccia.

```
    public bool MoveNext()
    {
        contaSequenza++;
        if (contaSequenza >= lunghezza)
            return false;  // la sequenza è terminata
        return true;
    }

    object System.Collections.IEnumerator.Current { get { return Current; } }
    public void Dispose() { }
    public void Reset() { }
}
```

Segue un frammento di codice che usa l'iteratore:

```
static void Main(string[] args)
{
    var it = new GeneratoreRnd(8, 1, 6);  // lunghezza:8  da:1  a:6
    while (it.MoveNext())
    {
        Console.WriteLine(it.Current);  // -> 4, 1, 1, 2, 2, 6, 5, 3
    }
}
```

2.2 Interfaccia IEnumerable<T>: uso del ciclo foreach

Un oggetto `GeneratoreRnd` è in grado di produrre una sequenza di numeri, *ma di per sé non è una sequenza di numeri*. Per questo motivo non è possibile utilizzare il ciclo `foreach`:

```
var it = new GeneratoreRnd(8, 1, 6);  // lunghezza:8  da:1  a:6
foreach (var numero in it)   // errore: foreach richiede una sequenza e non un iteratore!
{
    Console.WriteLine(numero);
}
...
```

In .NET una sequenza di oggetti è definita dall'interfaccia `IEnumerable<T>`, dove `T` rappresenta il tipo degli elementi della sequenza:

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
    IEnumerator GetEnumerator();        // ereditato da IEnumerable (obsoleto)
}
```

Nota bene: l'interfaccia definisce un solo metodo, `GetEnumerator()`, che ha la funzione di creare un iteratore in grado di produrre gli elementi della sequenza.

2.2.1 Implementazione di IEnumerable<T>

Di seguito implemento una classe che rappresenta una sequenza di numeri interi casuali:

```
public class SequenzaCasuale: IEnumerable<int>
{
    private int da, a, lunghezza;
    public SequenzaCasuale(int lunghezza, int da, int a)
    {
        this.da = da;
        this.a = a;
        this.lunghezza = lunghezza;
    }

    public IEnumerator<int> GetEnumerator()
    {
        return new GeneratoreRnd(lunghezza, da, a);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

Nota bene: `GetEnumerator()` crea l'iteratore `GeneratoreRnd` e lo restituisce al chiamante, il quale lo utilizzerà per ottenere i numeri della *sequenza*.

Ora è possibile utilizzare un `foreach` per scorrere i valori della *sequenza*:

```
var sequenza = new SequenzaCasuale(8, 1, 6);  // lunghezza:8  da:1  a:6
foreach (var numero in sequenza)
{
    Console.WriteLine(numero);  // -> 6, 6, 1, 2, 3, 4, 2, 4
}
```

2.2.2 Funzionamento del ciclo foreach

Il ciclo `foreach` non aggiunge niente all'uso degli iteratori mostrato in precedenza; il compilatore traduce `foreach` nell'uso del metodo `MoveNext()` e della proprietà `Current`:

Uso del ciclo foreach

```
var sequenza = new SequenzaCasuale(...);

foreach (var numero in sequenza)
{
    Console.WriteLine(numero);
}
```

Codice equivalente prodotto da C#

```
var sequenza = new SequenzaCasuale(...);

var it = sequenza.GetEnumerator();
while (it.MoveNext())
{
    Console.WriteLine(it.Current);
}
```

Naturalmente, nulla impedisce scrivere il codice di destra, ma non c'è convenienza nel farlo.

3 Sequenze ottenute da collezioni

Dal punto di vista del codice che lo usa, cosa vi sia dietro un iteratore non è rilevante. Per ottenere una sequenza degli oggetti memorizzati in una collezione occorre implementare un iteratore che restituisca gli elementi della collezione uno per volta.

3.1 Implementare IEnumerable<T> mediante una collezione

Supponiamo di voler gestire gli animali ricoverati in un ambulatorio veterinario. Esistono due tipi, `Animale` e `AnimaliRicoverati`; il secondo memorizza una lista di oggetti del primo tipo:

```
public class AnimaliRicoverati
{
    private List<Animale> animali = new List<Animale>();
    public void Aggiungi(Animale a) { animali.Add(a); }
    ...
}

public class Animale
{
    public string Nome { get; set; }
    public int Età { get; set; }
    public string Specie { get; set; }
}
```

`AnimaliRicoverati` memorizza una collezione di animali, *ma non è una collezione*, e non implementa `IEnumerable<>`. Se vogliamo utilizzarla in un `foreach`, dobbiamo realizzare un iteratore che restituisca gli elementi memorizzati in `animali`.

3.1.1 Iterare una collezione

L'iteratore scorrerà la lista animali, restituendo l'iesimo elemento mediante `Current`. L'indice dell'elemento corrente viene incrementato a ogni `MoveNext()`:

```
public class AnimaliIterator: IEnumerator<Animale>
{
    private List<Animale> animali;
    private int indice = -1;
    public AnimaliIterator(List<Animale> animali)
    {
        this.animali = animali;
    }

    public Animale Current
    {
        get { return animali[indice]; }
    }

    public bool MoveNext()
    {
        indice++;
        if (indice >= animali.Count)
```

```
            return false;
        return true;
    }

    public void Dispose(){}
    object IEnumerator.Current {get { return Current; }}
    public void Reset(){}
}
```

3.1.2 Implementare IEnumerable<T>

L'implementazione, al solito, si traduce nella definizione del metodo `GetEnumerator()`:

```
public class AnimaliRicoverati: IEnumerable<Animale>
{
    private List<Animale> animali = new List<Animale>();
    public void Aggiungi(Animale a) { animali.Add(a); }
    public void Rimuovi() {...}

    public IEnumerator<Animale> GetEnumerator()
    {
        return new AnimaliIterator(animali);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

Ora è possibile scrivere il seguente codice:

```
var animaliRicoverati = new AnimaliRicoverati();
animali.Aggiungi(new Animale { Nome = "Lampo", Specie = "Cane", Età = 10 });
animali.Aggiungi(new Animale { Nome = "Rosco", Specie = "Cane", Età = 4 });
animali.Aggiungi(new Animale { Nome = "Black", Specie = "Cane", Età = 2 });
animali.Aggiungi(new Animale { Nome = "Felix", Specie = "Gatto", Età = 12 });

foreach (var animale in animaliRicoverati)
{
    Console.WriteLine(animale.Nome);
}
```

3.2 Utilizzare l'iteratore definito da List<T>

L'esempio precedente è soltanto dimostrativo, poiché in realtà non c'è alcun bisogno di scrivere la classe `AnimaleIterator`: basta utilizzare l'iteratore già implementato da `List<>`:

```
public class AnimaliRicoverati: IEnumerable<Animale>
{
    private List<Animale> animali = new List<Animale>();
    ...
```

```
    public IEnumerator<Animale> GetEnumerator()
    {
        return animali.GetEnumerator();
    }
    ...
}
```

È uno scenario comune, poiché tutte le collezioni implementano l'interfaccia `IEnumerable<>` e dunque definiscono il metodo `GetEnumerator()`. D'altra parte, se si vuole restituire gli elementi in una sequenza diversa (invertita, ad esempio), occorre implementare un iteratore che lo faccia. C# fornisce una scorciatoia che evita la necessità di scrivere una classe come ho fatto finora.

4 Implementare iteratori mediante “yield return”

Gli esempi forniti finora sono serviti a svelare il funzionamento degli iteratori e dell'interfaccia `IEnumerator<>`, ma in realtà non c'è bisogno di creare una classe per implementare un iteratore, per due motivi:

- se gli elementi della sequenza provengono da una collezione, si può utilizzare l'iteratore di quest'ultima;
- si può implementare un iteratore utilizzando il costrutto `yield return`.

4.1 Implementare un iteratore mediante *yield return*

Il costrutto `yield return`, introdotto in C# 2.0, fornisce una comoda scorciatoia per implementare gli iteratori. Consideriamo nuovamente la classe `SequenzaCasuale`: questa restituisce un iteratore implementato dalla classe `GeneratoreRnd`. Ebbene, quest'ultima è inutile, poiché è possibile implementare l'iteratore mediante `yield return`:

```
public class SequenzaCasuale: IEnumerable<int>
{
    private int da, a, lunghezza;
    private Random rnd = new Random();
    public SequenzaCasuale(int lunghezza, int da, int a)
    {
        this.da = da;
        this.a = a;
        this.lunghezza = lunghezza;
    }

    public IEnumerator<int> GetEnumerator()
    {
        for (int i = 0; i < lunghezza; i++)
        {
            yield return rnd.Next(da, a + 1);
        }
    }

    // il metodo seguente non è più necessario (e così anche GeneratoreRnd)
    //public IEnumerator<int> GetEnumerator()
    //{
    //    return new GeneratoreRnd(lunghezza, da, a);
    //}

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

4.1.1 Funzionamento di “yield return”

Il funzionamento di `yield return` non è semplice da comprendere; in sostanza impone a C# di generare un iteratore che restituisca tanti elementi quante sono le chiamate all'istruzione `yield return`.

Consideriamo il seguente codice:

```
public IEnumerator<int> GetEnumerator()
{
    for (int i = 0; i < lunghezza; i++)
    {
        yield return rnd.Next(da, a + 1);
    }
}
```

Istruisce C# a realizzare una classe che implementa `IEnumerator<int>`. Ogni esecuzione di `yield return` corrisponde, “dall'altra parte”, all'esecuzione di `MoveNext()`.

Proviamo adesso ad analizzare il suo comportamento:

Uso dell'iteratore

```
var sequenza = new SequenzaCasuale(...);

foreach (var numero in sequenza)
{
    Console.WriteLine(numero);
}
```

Iteratore

```
public IEnumerator<int> GetEnumerator()
{
    for (int i = 0; i < lunghezza; i++)
    {
        yield return rnd.Next(da, a + 1);
    }
}
```

L'esecuzione del codice produce quanto segue:[2]

1) viene creato un oggetto di tipo `SequenzaCasuale()`

2) viene invocato `GetEnumerator()`: ciò non provoca l'esecuzione del ciclo for, poiché questo ciclo non esiste! Al suo posto, C# ha creato una classe con `MoveNext()` e `Current`.

3) comincia foreach: viene invocato il “metodo fantasma” `MoveNext()`, e cioè viene eseguita la prima iterazione del ciclo for in `GetEnumerator()`.

4) L'esecuzione di `yield return` “sospende” l'esecuzione del metodo; il controllo ritorna al ciclo foreach, dove viene visualizzata la variabile `numero`, che conterrà il valore della proprietà “fantasma” `Current`.

5) l'esecuzione ritorna al punto 3)

6) il termine del ciclo for di `GetEnumerator()` corrisponde al valore `false` restituito dal metodo “fantasma” `MoveNext()`: il ciclo foreach termina.

2 In realtà, si tratta di una descrizione concettualmente corretta, ma non effettiva di ciò che viene realmente generato dal compilatore.

4.1.2 Uso di yield return fuori da un ciclo

`yield return` viene spesso impiegato in un ciclo, ma ciò non rappresenta un vincolo. Ad esempio, il seguente metodo crea un iteratore che restituisce una sequenza di 3 numeri:

```
public IEnumerator<int> GetEnumerator()
{
    yield return rnd.Next(da, a + 1);
    yield return rnd.Next(da, a + 1);
    yield return rnd.Next(da, a + 1);
}
```

Il numero di esecuzioni di `yield return` determina il numero di elementi della sequenza.

4.2 Usare “yield return” per implementare IEnumerable<T>

Il costrutto `yield return` è in grado di restituire direttamente un oggetto di tipo **IEnumerable<>**. Basta che il metodo `GetEnumerator()` restituisca il tipo `IEnumerable<>`.

4.2.1 Produrre più sequenze dalla stessa collezione

È possibile usare `yield return` in qualsiasi metodo della stessa classe. Ciò consente di ottenere sequenze diverse a partire dalla stessa collezione di elementi. Consideriamo nuovamente la classe `AnimaliRicoverati`; può essere utile implementare un metodo che restituisca una sequenza di animali basata sulla specie:

```
public class AnimaliRicoverati: IEnumerable<Animale>
{
    private List<Animale> animali = new List<Animale>();
    public void Aggiungi(Animale a) { animali.Add(a); }
    public void Rimuovi() {...}


    public IEnumerator<Animale> GetEnumerator()
    {
        return new AnimaliIterator(animali);
    }

    public IEnumerable<Animale> AnimaliSpecie(string specie)
    {
        foreach (var a in animali)
        {
            if (a.Specie == specie)
                yield return a;
        }
    }

    object System.Collections.IEnumerator GetEnumerator() {return GetEnumerator();}
}
```

Il codice seguente mostra il funzionamento del metodo:

```
AnimaliRicoverati animali = new AnimaliRicoverati();
animali.Aggiungi(new Animale { Nome = "Lampo", Specie = "Cane", Età = 10 });
animali.Aggiungi(new Animale { Nome = "Rosco", Specie = "Cane", Età = 4 });
animali.Aggiungi(new Animale { Nome = "Black", Specie = "Cane", Età = 2 });
animali.Aggiungi(new Animale { Nome = "Felix", Specie = "Gatto", Età = 12 });

foreach (var a in animali)  // -> usa automaticamente GetEnumerator()
{
    Console.WriteLine(a.Nome);   // -> Lampo, Rosco, Black, Felix
}

foreach (var a in animali.AnimaliSpecie("Gatto"))
{
    Console.WriteLine(a.Nome);   // -> Felix
}
```

Nota bene: è fondamentale comprendere che il metodo `AnimaliSpecie()` non restituisce una collezione di animali, ma un *sequenza* di animali, ottenuti dalla collezione in base alla specie.

4.2.2 Generare una sequenza di numeri casuali con un metodo

Non è necessario creare una classe per produrre una sequenza, poiché è possibile svolgere tutto il lavoro in un singolo metodo:

```
static IEnumerable<int> GeneraSequenza(int lunghezza, int da, int a)
{
    Random rnd = new Random();
    for (int i = 0; i < lunghezza; i++)
    {
        yield return rnd.Next(da, a + 1);
    }
}
```

Dato che il metodo restituisce un oggetto di tipo `IEnumerable<int>`, è possibile utilizzarlo con un `foreach`:

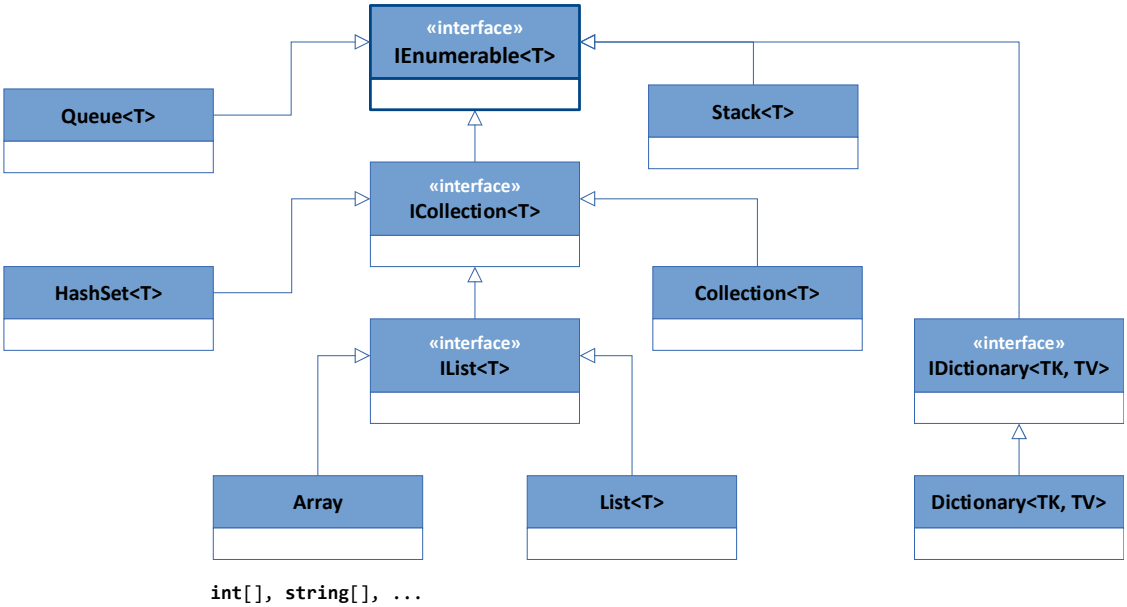
```
foreach (var numero in GeneraSequenza(8, 1, 6))
{
    Console.WriteLine(numero);  // -> 1, 2, 1, 4, 3, 6, 6, 4
}
```

5 Uso di IEnumerable<T>

`IEnumerable<>` è uno dei tipi più importanti e utilizzati in .NET. Alla base della sua importanza ci sono due idee molto semplici:

1. molti scenari richiedono di elaborare una sequenza di oggetti dello stesso tipo;
2. l'elaborazione non dipende dall'origine/organizzazione degli oggetti.

Il concetto di *sequenza di oggetti* è talmente importante che tutti i tipi che gestiscono un insieme di oggetti implementano l'interfaccia `IEnumerable<>`. La figura seguente fornisce una panoramica (incompleta):



Lo schema mostra che qualsiasi struttura dati omogenea è utilizzabile come una sequenza di elementi, indipendentemente dalla sua organizzazione interna.

Di seguito mi occuperò degli scenari nei quali occorre restituire una sequenza di oggetti.

5.1 Restituire una sequenza di elementi

Quando si implementa un metodo che restituisce un elenco di oggetti è necessario porsi due questioni:

1. stabilire il tipo restituito dal metodo: sequenza, lista, vettore, etc;
2. stabilire se restituire una collezione o utilizzare un iteratore.

In entrambi i casi non esiste una risposta sempre valida. Di seguito prendo in considerazione uno scenario e considero le varie possibilità.

5.1.1 Restituire un elenco di stringhe

Ipotizziamo di realizzare un metodo che legge un file di testo e ne restituisce il contenuto:

```
static string[] LeggiNominativi(string nomeFile)
{
    return File.ReadAllLines(nomeFile)
}
```

Poiché `ReadAllLines()` restituisce un vettore, è comodo utilizzare `string[]` come tipo restituito, ma ci si può porre la domanda: sarebbe opportuno restituire il tipo `IEnumerabe<string>`?

```
static IEnumerable<string> LeggiNominativi(string nomeFile)
{
    return File.ReadAllLines(nomeFile);
}
```

La risposta è: non ci sarebbero vantaggi. Infatti, `string[]` *è già un* `IEnumerable<string>`. Restituendo `string[]` si da al codice chiamante la possibilità di utilizzare l'elenco sia come vettore, che come *sequenza*:

```
IEnumerable<string> sequenza = LeggiNominativi("Nomi.txt");
string[] vettore = LeggiNominativi("Nomi.txt");
```

In conclusione: se un metodo memorizza gli elementi da restituire in una collezione è opportuno che il tipo restituito del metodo sia quello della collezione[3].

5.1.2 Restituire un iteratore

Quando un metodo restituisce un `IEnumerable<>`, il chiamante può utilizzare un `foreach` indipendentemente dall'organizzazione e/o origine dei dati. Ma il modo in cui i dati vengono prodotti può cambiare l'efficienza dell'intero processo.

Consideriamo nuovamente `LeggiNominativi()`: il metodo carica in memoria l'intero contenuto del file; non è un procedimento efficiente se il file contiene migliaia di righe e ciò che serve al chiamante è semplicemente accedere ad esse una per volta. Un'alternativa è quella di implementare un iteratore: il metodo leggerà una riga e la restituirà al chiamante prima di leggere la riga successiva. Così facendo:

- si evita di creare inutilmente una collezione di oggetti;
- si rendono immediatamente disponibili le righe del file man mano che vengono caricate, in modo che siano immediatamente processate dal chiamante.

Ecco la nuova versione del metodo, che implementa un iteratore:

```
static IEnumerable<string> LeggiNominativi(string nomeFile)
{
    using (var sr = new StreamReader(nomeFile))  // using garantisce che "sr" venga chiuso
    {
        var s = sr.ReadLine();
        while (s != null)
```

3 Esistono comunque altre considerazioni. Utilizzando **IEnumerable<string>**, saremmo in grado di modificare in seguito il corpo del metodo, senza essere costretti a cambiare l'intestazione. (Cosa che costringerebbe a modificare anche il codice chiamante.)

```
        {
            yield return s;   // "s" viene immediatamente elaborata dal chiamante
            s = sr.ReadLine();
        }
    }
}
```

Naturalmente, poiché viene restituito un iteratore, l'unica operazione concessa al chiamante è scorrere gli elementi della sequenza:

```
var righeFile = LeggiNominativi("Nomi.txt");
foreach (var riga in righeFile)
{
    Console.WriteLine(riga); // viene eseguita dopo ogni "yield return s"
}
```

5.1.3 Accesso "ritardato" agli elementi della sequenza: lazy loading

È importante non dimenticare mai che un iteratore restituisce gli elementi di una *sequenza soltanto quando vengono richiesti*; a questo riguardo si usa il termine *lazy loading*. Ciò ha ha un'implicazione fondamentale, che spesso si tende a ignorare.

Ad esempio, si consideri il seguente codice, che riutilizza il metodo `LeggiNominativi()`:

```
static void Main(string[] args)
{
    var nominativi = LeggiNominativi("Nomi.txt"); //->NON esegue iteratore
    Visualizza(nominativi); //->esegue iteratore (metodo LeggiNominativi())
    Visualizza(nominativi); //->esegue iteratore (metodo LeggiNominativi())

    var nomiPerA = nominativi.Where(n => n.StartsWith("A")); //->NON esegue iteratore
    Visualizza(nominativi); //->esegue iteratore (metodo LeggiNominativi())
}

static IEnumerable<string> LeggiNominativi(string nomeFile) {...}

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

La variabile `nominativi` non contiene affatto dei nomi, ma un iteratore che consente di accedere ai dati prodotti nel ciclo `while` del metodo `LeggiNominativi()`. Quando la si utilizza mediante un `foreach`, l'iteratore viene eseguito; nell'esempio, il metodo `LeggiNominativi()` viene eseguito tre volte e, paradossalmente, non quando viene effettivamente invocato.

Quello mostrato nell'esempio *non è* un uso intelligente degli iteratori; Nel caso sia necessario processare più volte gli elementi di una sequenza, si dovrebbe collezionarli in un vettore o una lista, utilizzando i metodi di estensione `ToArray()` o `ToList()`:

```
static void Main(string[] args)
{
    var nominativi = LeggiNominativi("Nomi.txt").ToArray();
    //-> ESEGUE iteratore e colleziona gli elementi nel vettore "nominativi"

    Visualizza(nominativi); //->scorre gli elementi del vettore "nominativi"
    ...
}
```

5.1.4 Conclusioni

Non c'è un criterio generale che stabilisca come implementare metodi che restituiscono insiemi di oggetti; dipende dai requisiti da soddisfare. In linea di massima, comunque, si dovrebbe optare per la restituzione di una sequenza mediante un iteratore. In questo modo si evita di creare anticipatamente collezioni di oggetti, lasciando che sia il codice chiamante a farlo, utilizzando `ToArray()` o `ToList()`.