

Capitolo 2

Fondamenti della sintassi LINQ

Language Integrated Query (LINQ) permette agli sviluppatori di interrogare e gestire sequenze di elementi (oggetti, entità, record di database, nodi XML, e via elencando) nelle proprie soluzioni software utilizzando una sintassi comune e un particolare linguaggio di programmazione indipendentemente dalla natura degli elementi gestiti. La caratteristica principale di LINQ è l'integrazione con i linguaggi di programmazione ampiamente utilizzati, un'integrazione resa possibile dall'utilizzo di una sintassi comune per tutti i tipi di contenuto.

Come abbiamo descritto nel Capitolo 1 "Introduzione a LINQ", LINQ fornisce un'infrastruttura di base per molte implementazioni differenti di engine di interrogazione, compreso LINQ to Objects, LINQ to SQL, LINQ to DataSet, LINQ to Entities, LINQ to XML, e via dicendo. Tutte queste estensioni di query si basano su *extension method* specializzati e condividono un insieme comune di parole chiave per la sintassi della espressione query che tratteremo in questo capitolo.

Prima di analizzare ciascuna parola chiave in dettaglio, analizzeremo vari aspetti di una semplice query LINQ e introdurremo il lettore agli elementi fondamentali della sintassi LINQ.

Query LINQ

LINQ si basa su un insieme di operatori query, definiti come *extension method*, che operano con qualsiasi oggetto che implementa l'interfaccia *IEnumerable<T>* o l'interfaccia *IQueryable<T>*.



Ulteriori informazioni Per ulteriori dettagli sugli *extension method*, si veda l'Appendice B "C# 3.0: nuove caratteristiche del linguaggio", e l'Appendice C "Visual Basic 2008: nuove caratteristiche del linguaggio".

Questo approccio rende LINQ un framework di interrogazione general-purpose poiché molte collection o tipi implementano *IEnumerable<T>* o *IQueryable<T>* e uno sviluppatore può definire la propria implementazione. Questa infrastruttura di interrogazione è anche altamente estensibile, come si vedrà nel Capitolo 12 "Estendere LINQ". Data l'architettura degli *extension method*, gli sviluppatori possono specializzare il comportamento di un metodo in base al tipo di dati che stanno interrogando. Ad esempio, sia LINQ to SQL sia LINQ to XML hanno operatori LINQ specializzati per gestire, rispettivamente, dati relazionali e nodi XML.

Sintassi delle query

Per introdurre la sintassi delle query, partiremo da un semplice esempio. Si immagini di dover interrogare un array di oggetti di un tipo *Developer* utilizzando LINQ to Objects e di estrarre i nomi degli sviluppatori che utilizzano C# come proprio principale linguaggio di programmazione. Il codice che si potrebbe utilizzare è mostrato nel Listato 2-1.

Listato 2-1 Una semplice espressione query in C# 3.0

```
using System;
using System.Linq;
using System.Collections.Generic;

public class Developer {
    public string Name;
    public string Language;
    public int Age;
}

class App {
    static void Main() {

        Developer[] developers = new Developer[] {
            new Developer {Name = "Paolo", Language = "C#"},
            new Developer {Name = "Marco", Language = "C#"},
            new Developer {Name = "Frank", Language = "VB.NET"}
        };

        var developersUsingCSharp =
            from d in developers
            where d.Language == "C#"
            select d.Name;

        foreach (var item in developersUsingCSharp) {
            Console.WriteLine(item);
        }
    }
}
```

Il risultato dell'esecuzione di questo codice sono i nomi *Paolo e Marco*.

In Visual Basic 2008, la stessa query, rispetto allo stesso tipo *Developer*, può essere espressa con una sintassi come quella mostrata nel Listato 2-2.

Listato 2-2 Una semplice espressione query in Visual Basic 2008

```
Imports System
Imports System.Linq
Imports System.Collections.Generic

Public Class Developer
    Public Name As String
    Public Language As String
```

```

    Public Age As Integer
End Class
Module App
    Sub Main()
        Dim developers As New Developer() { _
            New Developer With {.Name = "Paolo", .Language = "C#"}, _
            New Developer With {.Name = "Marco", .Language = "C#"}, _
            New Developer With {.Name = "Frank", .Language = "VB.NET"}}

        Dim developersUsingCSharp = _
            From d In developers _
            Where d.Language = "C#" _
            Select d.Name

        For Each item in developersUsingCSharp
            Console.WriteLine(item)
        Next
    End Sub
End Module

```

La sintassi delle query (mostrata in grassetto nel Listato 2-1 e nel Listato 2-2) è detta *espressione query*. In alcune implementazioni LINQ, una rappresentazione in memoria di queste query è nota come *albero di espressioni*. Una espressione query opera su una o ulteriori sorgenti di informazioni applicando uno o più operatori query presenti nel gruppo degli operatori query standard o degli operatori specifici al dominio. In generale, la valutazione di una espressione query produce una sequenza di valori. Una espressione query viene valutata solo quando viene enumerato il relativo contenuto. Per ulteriori dettagli sulle espressioni query e sugli alberi delle espressioni, si consulti il Capitolo 11, "All'interno degli alberi di espressioni".



Nota Per semplicità, negli esempi seguenti tratteremo solo la sintassi C# 3.0; tuttavia, si può capire che la versione Visual Basic 2008 di questi esempi è molto simile a quella C# 3.0.

Queste query si leggono come qualcosa di simile a una istruzione SQL, benché abbiano uno stile un po' differente. L'espressione d'esempio che abbiamo definito consiste di un comando di selezione:

```
select d.Name
```

applicato a un insieme di elementi:

```
from d in developers
```

dove la clausola *from* punta a una istanza di una classe che implementa l'interfaccia *IEnumerable<T>*. La selezione si applica a una specifica condizione di filtraggio:

```
where d.Language == "C#"
```

Queste clausole vengono tradotte dai compilatori dei linguaggi in invocazioni degli *extension method* che vengono applicati sequenzialmente alla destinazione della

query. La libreria principale di LINQ, definita nell'assembly System.Core.dll, definisce un insieme di *extension method* raggruppati per destinazione e scopo. Ad esempio, l'assembly comprende una classe di nome *Enumerable*, definita nel namespace *System.Linq*, che definisce gli *extension method* che possono essere applicati alle istanze dei tipi che implementano l'interfaccia *IEnumerable<T>*.

La condizione di filtro (*where*) definita nella query di esempio si traduce in una invocazione del metodo di estensione *Where* della classe *Enumerable*. Questo metodo fornisce due overload, entrambi i quali accettano una delegate a una funzione *predicato* che descrive la condizione filtro da verificare mentre partiziona i dati risultanti. In questo caso, il *predicato* di filtro è una delegate generica che accetta un elemento di tipo *T*, che è lo stesso tipo delle istanze memorizzate nell'enumerazione che stiamo filtrando. La delegate restituisce un risultato *Boolean* che indica lo stato di "appartenenza" dell'elemento nel resultset filtrato.

```
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Come si può vedere dalla firma del metodo, si può invocare questo metodo per qualsiasi tipo che implementi *IEnumerable<T>*; perciò, possiamo invocarlo sull'array *developers* come segue:

```
var filteredDevelopers = developers.Where(delegate (Developer d) {
    return (d.Language == "C#");
});
```

Qui l'argomento *predicato*, passato al metodo *Where*, rappresenta una delegate anonima a una funzione che viene invocata per ciascun elemento di tipo *Developer* estratto dall'insieme sorgente dei dati (*developers*). Il risultato della invocazione del metodo *Where* sarà un sottoinsieme degli elementi: tutti quelli che verificano la condizione *predicato*.

In C# 3.0 e in Visual Basic 2008, una delegate anonima può essere definita in un modo più facile, utilizzando una espressione lambda. Utilizzando una espressione lambda, il codice di filtraggio d'esempio può essere riscritto in un modo più compatto:

```
var filteredDevelopers = developers.Where(d => d.Language == "C#");
```



Importante Per ulteriori dettagli sulla sintassi degli *extension method*, delle espressioni lambda, delle delegate anonime, e via elencando si consulti l'Appendice B e l'Appendice C.

Lo statement *select* è anche un metodo di estensione (denominato *Select*) fornito dalla classe *Enumerable*. Ecco la signature del metodo *Select*:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector);
```

L'argomento *selector* è una proiezione che restituisce una enumerazione di oggetti di tipo *TResult*, ottenuta da un insieme di oggetti sorgenti di tipo *TSource*. Come abbiamo fatto in precedenza, possiamo applicare questo metodo all'intera collection *developers* utilizzando un'espressione lambda. O possiamo invocarlo sulla collection filtrata dal linguaggio di programmazione (denominata *filteredDevelopers*) poiché si tratta ancora di un tipo che implementa *IEnumerable<T>*:

```
var csharpDevelopersNames = filteredDevelopers.Select(d => d.Name);
```

In base alla sequenza di istruzioni che abbiamo appena descritto, possiamo riscrivere la query di esempio senza utilizzare la sintassi dell'espressione query:

```
IEnumerable<string> developersUsingCSharp =
    developers
    .Where(d => d.Language == "C#")
    .Select(d => d.Name);
```

Entrambi i metodi *Where* e *Select* ricevono espressioni lambda come argomenti. Queste espressioni lambda si traducono in predicati e proiezioni che si basano su un insieme di tipi delegate generic definito nel namespace *System*, nell'assembly *System.Core.dll*.

Ecco l'intera famiglia di tipi delegate generic disponibili. Molti extension method della classe *Enumerable* accettano queste delegate come argomenti, e le utilizzeremo in tutti gli esempi di questo capitolo.

```
public delegate TResult Func< TResult >();
public delegate TResult Func< T, TResult >( T arg );
public delegate TResult Func< T1, T2, TResult >( T1 arg1, T2 arg2 );
public delegate TResult Func< T1, T2, T3, TResult >
    ( T1 arg1, T2 arg2, T3 arg3 );
public delegate TResult Func< T1, T2, T3, T4, TResult >
    ( T1 arg1, T2 arg2, T3 arg3, T4 arg4 );
```

Una versione finale della nostra query iniziale potrebbe essere simile a quella del Listato 2-3.

Listato 2-3 La prima espressione query tradotta in elementi di base

```
Func<Developer, bool> filteringPredicate = d => d.Language == "C#";
Func<Developer, string> selectionPredicate = d => d.Name;
IEnumerable<string> developersUsingCSharp =
    developers
    .Where(filteringPredicate)
    .Select(selectionPredicate);
```

Il compilatore C# 3.0, così come il compilatore Visual Basic 2008, traduce le espressioni query LINQ (Listato 2-1 e Listato 2-2) in qualcosa di simile allo statement mostrato nel Listato 2-3. Dopo aver acquisito familiarità con la sintassi delle espressioni query (Listato 2-1 e Listato 2-2), è più semplice e più facile scrivere e gestire questa sintassi, anche se è opzionale e si può sempre utilizzare la versione equivalente, più prolissa (Listato 2-3). Nondimeno, talvolta è necessario utilizzare l'invocazione diretta a un metodo di estensione poiché la sintassi della espressione query non copre tutti i possibili *extension method*.



Importante Nel Capitolo 3 “LINQ to Objects”, tratteremo in maggior dettaglio tutti gli extension method disponibili nella classe *Enumerable* definita nel namespace *System.Linq*.

Sintassi completa delle query

Nei paragrafi precedenti, abbiamo descritto una semplice query su un elenco di oggetti. La sintassi dell'espressione query, tuttavia, è più completa e articolata di quella mostrata in quell'esempio, e fornisce molte differenti parole chiave del linguaggio che soddisfano gran parte dei comuni scenari di interrogazione. Ogni query inizia con una clausola *from* e termina con una clausola *select* o con una clausola *group*. Il motivo di far iniziare la query con una clausola *from* invece di un'istruzione *select*, come avviene nella sintassi SQL, ha a che fare (tra gli altri motivi tecnici) con la necessità di fornire funzionalità Microsoft IntelliSense nella parte restante della query, che rende più facile la scrittura delle condizioni, delle selezioni, e di ogni altra clausola dell'espressione query. Una clausola *select* proietta il risultato di un'espressione in un oggetto enumerabile. Una clausola *group* proietta il risultato di un'espressione in un insieme di gruppi, in base alla condizione di raggruppamento, dove ciascun gruppo è un oggetto enumerabile. Il seguente codice mostra un prototipo della sintassi completa di un'espressione query:

```
query-expression ::= from-clause query-body

query-body ::=
join-clause*
(from-clause join-clause* | let-clause | where-clause)*
orderby-clause?
(select-clause | groupby-clause)

query-continuation?

from-clause ::= from itemName in srcExpr

select-clause ::= select selExpr

groupby-clause ::= group selExpr by keyExpr
```

La prima clausola *from* può essere seguita da zero o più clausole *from*, *let* o *where*. Una clausola *let* applica un nome al risultato di un'espressione; è utile ogni qualvolta è necessario referenziare la stessa espressione molte volte in una query.

```
let-clause ::= let itemName = selExpr
```

Una clausola *where*, come abbiamo già detto, definisce un filtro che viene applicato per includere specifici elementi nei risultati.

```
where-clause ::= where predExpr
```

Ciascuna clausola *from* genera una “variabile intervallo” (*range variable*) locale che corrisponde a ciascun elemento nella sequenza sorgente su cui vengono applicati gli

operatori query (come i metodi di estensione di *System.Linq.Enumerable*).

Una clausola *from* può essere seguita da qualsiasi numero di clausole *join*. La clausola finale *select* o *group* può essere preceduta da una clausola *orderby* che applica un ordinamento ai risultati:

```
join-clause ::=
join itemName in srcExpr on keyExpr equals keyExpr
(into itemName)?

orderby-clause ::= orderby (keyExpr (ascending | descending)?) *

query-continuation ::= into itemName query-body
```

Si vedranno esempi di espressioni query in tutto questo libro. Quando si vogliono controllare elementi specifici della loro sintassi si potrà far riferimento a questi paragrafi.

Parole chiave della query

Nei prossimi paragrafi, descriveremo in maggior dettaglio le varie parole chiave della query disponibili nella sintassi dell'espressione query.

Clausola *from*

La prima parola chiave è la clausola *from*. Questa definisce la fonte dati di una query o di una sottoquery e una variabile “range” (ossia “intervallo”) che definisce ciascun singolo elemento da interrogare dal datasource. Il datasource può essere qualsiasi istanza di un tipo che implementa le interfacce *IEnumerable*, *IEnumerable<T>* o *IQueryable<T>*, che implementa *IEnumerable<T>*. Nel seguente estratto, si può vedere un esempio di frase C# 3.0 che utilizza questa clausola:

```
from rangeVariable in dataSource
```

Il compilatore del linguaggio inferisce il tipo della variabile range dal tipo del datasource. Ad esempio, se il datasource è di tipo *IEnumerable<Developer>*, la variabile range sarà di tipo *Developer*. Nei casi in cui non si utilizza un datasource fortemente tipizzato (ad esempio un *ArrayList* di oggetti di tipo *Developer* che implementa *IEnumerable*) si dovrà fornire esplicitamente il tipo della variabile range. Nel Listato 2-4, si può vedere un esempio di una siffatta query con una dichiarazione esplicita del tipo *Developer* della variabile range denominata *d*.

Listato 2-4 Una espressione query su un datasource non generic, con la dichiarazione del tipo della variabile range

```
ArrayList developers = new ArrayList();
developers.Add(new Developer { Name = "Paolo", Language = "C#" });
developers.Add(new Developer { Name = "Marco", Language = "C#" });
developers.Add(new Developer { Name = "Frank", Language = "VB.NET" });
```

```

var developersUsingCSharp =
    from Developer d in developers
    where d.Language == "C#"
    select d.Name;

foreach (string item in developersUsingCSharp) {
    Console.WriteLine(item);
}

```

Nel precedente esempio, il casting è obbligatorio; altrimenti, la query non verrà compilata poiché il compilatore non può desumere automaticamente il tipo della variabile range, perdendo perciò la capacità di risolvere l'accesso ai membri *Language* e *Name* della query stessa.

Le query possono avere più clausole *from* per poter definire i join tra più datasource. In C# 3.0, ciascun datasource richiede la dichiarazione di una clausola *from*, come si può vedere nel Listato 2-5, dove abbiamo messo in join i clienti con i relativi ordini. Si noti che la relazione tra *Customer* e *Order* è definita fisicamente dalla presenza di un array *Orders* di tipo *Order* in ciascuna istanza di *Customer*.



Importante Quando si utilizzano più clausole *from*, la “condizione di join” viene determinata dalla struttura dei dati ed è differente dal concetto di join in un database relazionale. (Per far questo, è necessario utilizzare la clausola *join* in una espressione query, aspetto che tratteremo in seguito in questo capitolo).

Listato 2-5 Una espressione query C# 3.0 con un join tra due datasource

```

public class Customer {
    public String Name { get; set; }
    public String City { get; set; }
    public Order[] Orders { get; set; }
}

public class Order {
    public Int32 IdOrder { get; set; }
    public Decimal EuroAmount { get; set; }
    public String Description { get; set; }
}

// ... codice omissso ...

static void queryWithJoin() {
    Customer[] customers = new Customer[] {
        new Customer { Name = "Paolo", City = "Brescia",
            Orders = new Order[] {
                new Order { IdOrder = 1, EuroAmount = 100, Description = "Order 1" },
                new Order { IdOrder = 2, EuroAmount = 150, Description = "Order 2" },
                new Order { IdOrder = 3, EuroAmount = 230, Description = "Order 3" },
            }},
        new Customer { Name = "Marco", City = "Torino",
            Orders = new Order[] {
                new Order { IdOrder = 4, EuroAmount = 320, Description = "Order 4" },
                new Order { IdOrder = 5, EuroAmount = 170, Description = "Order 5" },
            }},
    };
}

```

```

var ordersQuery =
    from c in customers
    from o in c.Orders
    select new { c.Name, o.IdOrder, o.EuroAmount };

foreach (var item in ordersQuery) {
    Console.WriteLine(item);
}
}

```

In Visual Basic 2008, una unica clausola *From* può definire più datasource, separati da virgole, come si può vedere nel Listato 2-6.

Listato 2-6 Una espressione query Visual Basic 2008 con un join tra due datasource

```

Dim customers As Customer() = { _
    New Customer With {.Name = "Paolo", .City = "Brescia", _
        .Orders = New Order() { _
            New Order With {.IdOrder = 1, .EuroAmount = 100, .Description = "Order 1"}, _
            New Order With {.IdOrder = 2, .EuroAmount = 150, .Description = "Order 2"}, _
            New Order With {.IdOrder = 3, .EuroAmount = 230, .Description = "Order 3"} _
        }}, _
    New Customer With {.Name = "Marco", .City = "Torino", _
        .Orders = New Order() { _
            New Order With {.IdOrder = 4, .EuroAmount = 320, .Description = "Order 4"}, _
            New Order With {.IdOrder = 5, .EuroAmount = 170, .Description = "Order 5"} _
        } _
    } _
}
Dim ordersQuery = _
    From c In customers, _
        o In c.Orders _
    Select c.Name, o.IdOrder, o.EuroAmount
For Each item In ordersQuery
    Console.WriteLine(item)
Next

```

Tratteremo i join in maggior dettaglio in seguito nel capitolo.

Clausola *where*

Come abbiamo già detto, la clausola *where* specifica una condizione di filtraggio da applicare al datasource. Il predicato applica una condizione Boolean a ciascun elemento nel datasource, estraendo solo quegli elementi che vengono valutati a *true*. In una singola query, si possono avere più clausole *where* o una clausola *where* con più predicati che vengono combinati utilizzando gli operatori logici (&&, ||, e ! in C# 3.0, o *And*, *Or*, *AndAlso*, *OrElse*, *Is* e *IsNot* in Visual Basic 2008). In Visual Basic 2008, il predicato può essere qualsiasi espressione equivalente a un valore *Boolean*, pertanto si può utilizzare anche un'espressione numerica che verrà considerata *true* se non è uguale a zero.

Si consideri la query nel Listato 2-7, in cui utilizziamo la clausola *where* per estrarre tutti gli ordini con *EuroAmount* maggiore di 200 Euro.

Listato 2-7 Un'espressione query C# 3.0 con una clausola *where*

```
var ordersQuery =
    from c in customers
    from o in c.Orders
    where o.EuroAmount > 200
    select new { c.Name, o.IdOrder, o.EuroAmount };
```

Nel Listato 2-8, si può vedere la sintassi della query corrispondente utilizzando Visual Basic 2008.

Listato 2-8 Un'espressione query Visual Basic 2008 con una clausola *where*

```
Dim ordersQuery = _
    From c In customers, _
        o In c.Orders _
    Where o.EuroAmount > 200 _
    Select c.Name, o.IdOrder, o.EuroAmount
```

Clausola *select*

La clausola *select* specifica la “forma” dell’output della query. Si basa su una proiezione che determina cosa selezionare dal risultato della valutazione di tutte le clausole e tutte le espressioni che la precedono. In Visual Basic 2008, la clausola *Select* non è obbligatoria. Se non viene specificata, la query restituisce un tipo che si basa sulla variabile range identificata per l’ambito di visibilità corrente. Nel Listato 2-7 e nel Listato 2-8, abbiamo utilizzato la clausola *select* per proiettare tipi anonimi costituiti da proprietà o membri delle variabili range “in scope”. Come si può vedere confrontando la sintassi C# 3.0 (Listato 2-7) e la sintassi Visual Basic 2008 (Listato 2-8), quest’ultima sembra più uno statement SQL nel pattern di selezione, mentre la prima appare più simile alla sintassi del linguaggio di programmazione. Infatti, in C# 3.0 è necessario dichiarare esplicitamente il proprio intento di creare una nuova istanza del tipo anonimo, mentre in Visual Basic 2008 la sintassi del linguaggio è più leggera e occulta il funzionamento interno.

Clausole *group* e *into*

La clausola *group* può essere utilizzata per proiettare un risultato raggruppato in base a una chiave. Può essere utilizzata come alternativa alla clausola *from* e permette di utilizzare chiavi a valore singolo così come chiavi con più valori. Nel Listato 2-9, si può vedere un esempio di query che raggruppa gli sviluppatori per linguaggio di programmazione.

Listato 2-9 Un'espressione query C# 3.0 per raggruppare gli sviluppatori per linguaggio di programmazione

```
Developer[] developers = new Developer[] {
    new Developer { Name = "Paolo", Language = "C#" },
    new Developer { Name = "Marco", Language = "C#" },
    new Developer { Name = "Frank", Language = "VB.NET" },
};

var developersGroupedByLanguage =
    from d in developers
    group d by d.Language;

foreach (var group in developersGroupedByLanguage) {
    Console.WriteLine("Language: {0}", group.Key);
    foreach (var item in group) {
        Console.WriteLine("\t{0}", item.Name);
    }
}
```

L'output del frammento di codice del Listato 2-9 è il seguente:

```
Language: C# Paolo Marco
Language: VB.NET Frank
```

Come si può vedere nell'esempio di codice, il risultato della query è una enumerazione dei gruppi identificati da una chiave ed è costituita da elementi interni. Infatti, enumeriamo ciascun gruppo del risultato della query, scrivendo la relativa proprietà *Key* sulla console e esaminando gli elementi in ciascun gruppo per estrarne i valori. Come abbiamo detto in precedenza, si possono raggruppare gli elementi utilizzando una chiave a più valori che si serve dei tipi anonimi. Un esempio è mostrato nel Listato 2-10, dove raggruppiamo gli sviluppatori per linguaggio e fascia di età.

Listato 2-10 Un'espressione query C# 3.0 per raggruppare gli sviluppatori per linguaggio di programmazione e fascia di età

```
Developer[] developers = new Developer[] {
    new Developer { Name = "Paolo", Language = "C#", Age = 32 },
    new Developer { Name = "Marco", Language = "C#", Age = 37 },
    new Developer { Name = "Frank", Language = "VB.NET", Age = 48 },
};

var developersGroupedByLanguage =
    from d in developers
    group d by new { d.Language, AgeCluster = (d.Age / 10) * 10 };

foreach (var group in developersGroupedByLanguage) {
    Console.WriteLine("Language: {0}", group.Key);
    foreach (var item in group) {
        Console.WriteLine("\t{0}", item.Name);
    }
}
```

Questa volta l'output del frammento di codice del Listato 2-10 è il seguente:

```
Language: { Language = C#, AgeCluster = 30 } Paolo Marco
Language: { Language = VB.NET, AgeCluster = 40 } Frank
```

In questo esempio, la chiave *Key* per ciascun gruppo è un tipo anonimo definito da due proprietà: *Language* e *AgeCluster*.

Visual Basic 2008 supporta anche il raggruppamento dei risultati utilizzando la clausola *Group By*. Nel Listato 2-11, si può vedere un esempio di una query che è equivalente a quella mostrata nel Listato 2-9.

Listato 2-11 Un'espressione query Visual Basic 2008 per raggruppare gli sviluppatori per linguaggio di programmazione

```
Dim developers As Developer() = { _
    New Developer With {.Name = "Paolo", .Language = "C#", .Age = 32}, _
    New Developer With {.Name = "Marco", .Language = "C#", .Age = 37}, _
    New Developer With {.Name = "Frank", .Language = "VB.NET", .Age = 48}}

Dim developersGroupedByLanguage = _
    From d In developers _
    Group d By d.Language Into Group _
    Select Language, Group

For Each group In developersGroupedByLanguage
    Console.WriteLine("Language: {0}", group.Language)
    For Each item In group.Group
        Console.WriteLine(" {0}", item.Name)
    Next
Next
```

La sintassi Visual Basic 2008 è un po' più complessa della corrispondente sintassi C# 3.0. In Visual Basic 2008, è necessario proiettare il raggruppamento utilizzando la clausola *Into* per creare un nuovo oggetto *Group* di elementi e poi dichiarare esplicitamente il pattern di selezione. Tuttavia, il risultato del raggruppamento è più facile da enumerare poiché il valore *Key* mantiene lo stesso nome (*Language*).

Anche C# 3.0 fornisce una clausola *into* che è utile in combinazione con la parola chiave *group*, anche se l'utilizzo non è obbligatorio. Si può utilizzare la parola chiave *into* per memorizzare i risultati di uno statement *select*, *group* o *join* in una variabile temporanea. Si potrebbe utilizzare questa costruzione quando è necessario eseguire ulteriori query sui risultati. A causa di questo comportamento, questa parola chiave è anche detta una clausola di *continuazione*. Nel Listato 2-12, si può vedere un esempio di una espressione query C# 3.0 che utilizza la clausola *into*.

Listato 2-12 Un'espressione query C# 3.0 che utilizza la clausola *into*

```
var developersGroupedByLanguage =
    from d in developers
    group d by d.Language into developersGrouped
    select new {
        Language = developersGrouped.Key,
        DevelopersCount = developersGrouped.Count()
    };

foreach (var group in developersGroupedByLanguage) {
    Console.WriteLine("Language {0} contains {1} developers",
        group.Language, group.DevelopersCount);
}
```

Clausola *orderby*

La clausola *orderby*, come si può desumere dal nome, permette di ordinare il risultato di una query in ordine crescente o decrescente. L'ordinamento può essere eseguito utilizzando una o più chiavi che combinano differenti direzioni di ordinamento. Il Listato 2-13 mostra un esempio di una query per estrarre gli ordini effettuati dai clienti, ordinati per *EuroAmount*.

Listato 2-13 Un'espressione query C# 3.0 con una clausola *orderby*

```
var ordersSortedByEuroAmount =
    from c in customers
    from o in c.Orders
    orderby o.EuroAmount
    select new { c.Name, o.IdOrder, o.EuroAmount };
```

Il Listato 2-14 mostra un esempio di query che seleziona gli ordini ordinati per nome cliente (*Name*) e *EuroAmount* in ordine decrescente.

Listato 2-14 Un'espressione query C# 3.0 con una clausola *orderby* con più condizioni di ordinamento

```
var ordersSortedByCustomerAndEuroAmount =
    from c in customers
    from o in c.Orders
    orderby c.Name, o.EuroAmount descending
    select new { c.Name, o.IdOrder, o.EuroAmount };
```

Nel Listato 2-15, si può vedere la query corrispondente scritta in Visual Basic 2008.

Listato 2-15 Un'espressione query Visual Basic 2008 con una clausola *orderby* con più condizioni di ordinamento

```
Dim ordersSortedByCustomerAndEuroAmount = _
    From c In customers, _
        o In c.Orders _
    Order By c.Name, o.EuroAmount Descending _
    Select c.Name, o.IdOrder, o.EuroAmount
```

Qui entrambi i linguaggi hanno una sintassi molto simile.

Clausola *join*

La parola chiave *join* permette di associare differenti datasource sulla base di un membro che può essere confrontato per equivalenza. Funziona in modo analogo a uno statement SQL di equijoin. Non si possono confrontare elementi da porre in join utilizzando confronti come "maggiore di", "minore di" o "diverso da". Si possono definire confronti di uguaglianza solo utilizzando una speciale parola chiave *equals* che

ha un comportamento differente dall'operatore `==` poiché la posizione degli operandi ha importanza. Con *equals*, la chiave sinistra consuma la sequenza sorgente esterna e la chiave destra consuma la sorgente interna. La sorgente esterna ha ambito di visibilità solo nella parte sinistra di *equals*, e la sequenza sorgente interna ha ambito di visibilità solo nella parte destra. Ecco questo concetto presentato in pseudocodice.

```
clausola-join ::= join elementoInterno in sequenzaInterna
                on chiaveEsterna equals chiaveInterna
```

Utilizzando la clausola *join*, si possono definire operazioni di *inner join*, *group join* e *left outer join*. Un "inner join" è un join che restituisce un risultato semplice mappando gli elementi del datasource esterno con il corrispondente datasource interno. Questa operazione salta gli elementi del datasource esterno privi di elementi corrispondenti nel datasource interno. Il Listato 2-16 presenta una semplice query con una inner join tra "categorie prodotti" e i relativi prodotti.

Listato 2-16 Un'espressione query C# 3.0 con un inner join

```
public class Category {
    public Int32 IdCategory { get; set; }
    public String Name { get; set; }
}

public class Product {
    public String IdProduct { get; set; }
    public Int32 IdCategory { get; set; }
    public String Description { get; set; }
}

// ... codice omissso ...

Category[] categories = new Category[] {
    new Category { IdCategory = 1, Name = "Pasta"},
    new Category { IdCategory = 2, Name = "Beverages"},
    new Category { IdCategory = 3, Name = "Other food"},
};

Product[] products = new Product[] {
    new Product { IdProduct = "PASTA01", IdCategory = 1, Description = "Tortellini" },
    new Product { IdProduct = "PASTA02", IdCategory = 1, Description = "Spaghetti" },
    new Product { IdProduct = "PASTA03", IdCategory = 1, Description = "Fusilli" },
    new Product { IdProduct = "BEV01", IdCategory = 2, Description = "Water" },
    new Product { IdProduct = "BEV02", IdCategory = 2, Description = "Orange Juice" },
};

var categoriesAndProducts =
    from c in categories
    join p in products on c.IdCategory equals p.IdCategory
    select new {
        c.IdCategory,
        CategoryName = c.Name,
        Product = p.Description
    };

foreach (var item in categoriesAndProducts) {
    Console.WriteLine(item);
}
```

L'output di questo estratto di codice è simile all'output sottostante. Si noti che la categoria "Other food" è assente poiché non contiene prodotti.

```
{ IdCategory = 1, CategoryName = Pasta, Product = Tortellini }
{ IdCategory = 1, CategoryName = Pasta, Product = Spaghetti }
{ IdCategory = 1, CategoryName = Pasta, Product = Fusilli }
{ IdCategory = 2, CategoryName = Beverages, Product = Water }
{ IdCategory = 2, CategoryName = Beverages, Product = Orange Juice }
```

Un *group join* ("join di gruppo") definisce un join che produce un resultset gerarchico, raggruppando gli elementi della sequenza interna con i relativi elementi corrispondenti della sequenza esterna. Nei casi in cui un elemento della sequenza esterna è privo dei corrispondenti elementi della sequenza interna, l'elemento esterno verrà messo in join con un array vuoto. Un *group join* non ha un equivalente relazionale nella sintassi SQL a causa del risultato gerarchico. Nel Listato 2-17, si può vedere un esempio di una siffatta query. (Si vedrà una forma espansa di questo tipo di query nel Capitolo 3).

Listato 2-17 Un'espressione query C# 3.0 con un *group join*

```
var categoriesAndProducts =
    from c in categories
    join p in products on c.IdCategory equals p.IdCategory
    into productsByCategory
    select new {
        c.IdCategory,
        CategoryName = c.Name,
        Products = productsByCategory
    };

foreach (var category in categoriesAndProducts) {
    Console.WriteLine("{0} -{1}", category.IdCategory, category.CategoryName);
    foreach (var product in category.Products) {
        Console.WriteLine("\t{0}", product.Description);
    }
}
```

Si noti che stavolta la categoria "Other food" è presente nell'output, anche se è vuota:

```
1 - Pasta
   Tortellini
   Spaghetti
   Fusilli
2 - Beverages
   Water
   Orange Juice
3 - Other food
```

Visual Basic 2008 fornisce una parola chiave specifica detta *Group Join* per definire join di gruppi nelle espressioni query.

Un *left outer join* ("join sinistro esterno") restituisce un resultset semplice che comprende ogni elemento della sorgente esterna anche se manca l'elemento

corrispondente della sorgente interna. Per produrre questo risultato, è necessario utilizzare il metodo di estensione *DefaultIfEmpty*, che restituisce un valore di default nel caso di un valore vuoto di datasource. Tratteremo questo e molti altri metodi di estensione in maggior dettaglio nel Capitolo 3. Nel Listato 2-18, si può vedere un esempio di questa sintassi.

Listato 2-18 Una espressione query C# 3.0 con un left outer join

```
var categoriesAndProducts =
    from c in categories
    join p in products on c.IdCategory equals p.IdCategory
    into productsByCategory
    from pc in productsByCategory.DefaultIfEmpty(
        new Product {
            IdProduct = String.Empty,
            Description = String.Empty,
            IdCategory = 0})
    select new {
        c.IdCategory,
        CategoryName = c.Name,
        Product = pc.Description
    };

foreach (var item in categoriesAndProducts) {
    Console.WriteLine(item);
}
```

Questo esempio produce il seguente output nella console:

```
{ IdCategory = 1, CategoryName = Pasta, Product = Tortellini }
{ IdCategory = 1, CategoryName = Pasta, Product = Spaghetti }
{ IdCategory = 1, CategoryName = Pasta, Product = Fusilli }
{ IdCategory = 2, CategoryName = Beverages, Product = Water }
{ IdCategory = 2, CategoryName = Beverages, Product = Orange Juice }
{ IdCategory = 3, CategoryName = Other food, Product = }
```

Si noti che la categoria “Other food” è presente con un prodotto vuoto, che è fornito dal metodo di estensione *DefaultIfEmpty*.

Un ultimo punto da sottolineare sulla clausola *join* è che si possono confrontare elementi utilizzando chiavi composte. Si fa semplicemente uso dei tipi anonimi come abbiamo mostrato con la parola chiave *group*. Ad esempio, se si avesse una chiave composta in *Category* costituita da *IdCategory* e *Year*, si potrebbe scrivere la seguente istruzione con un tipo anonimo utilizzato nella condizione *equals*:

```
from c in categories
join p in products
on new { c.IdCategory, c.Year } equals new { p.IdCategory, p.Year }
into productsByCategory
```

Come avete già visto in questo capitolo, si possono anche ottenere i risultati dei *join* utilizzando clausole *from* nidificate, il che è un utile approccio ogni qualvolta è necessario definire query non di *equijoin*.

Visual Basic 2008 ha una sintassi abbastanza simile a C# 3.0, ma offre anche alcune scorciatoie per definire i *join* più rapidamente. Possiamo definire istruzioni di *join*

implicite utilizzando più clausole *In* nello statement *From* e definendo le condizioni di uguaglianza con una clausola *Where*. Nel Listato 2-19, si può vedere un esempio di questa sintassi.

Listato 2-19 Un'istruzione di join implicito in Visual Basic 2008

```
Dim categoriesAndProducts = _
    From c In categories, p In products _
    Where c.IdCategory = p.IdCategory _
    Select c.IdCategory, CategoryName = c.Name, Product = p.Description

For Each item In categoriesAndProducts
    Console.WriteLine(item)
Next
```

Nel Listato 2-20, si può vedere la stessa query definita utilizzando la sintassi di *join* esplicito standard.

Listato 2-20 Un'istruzione di join esplicito in Visual Basic 2008

```
Dim categoriesAndProducts = _
    From c In categories Join p In products _
    On p.IdCategory Equals c.IdCategory _
    Select c.IdCategory, CategoryName = c.Name, Product = p.Description
```

Si noti che in Visual Basic 2008 l'ordine degli elementi nel confronto di uguaglianza non importa poiché il compilatore li organizzerà da sé, rendendo la sintassi della query meno rigida, come accade nel classico SQL relazionale.

Clausola *let*

La clausola *let* permette di memorizzare il risultato di una sottoespressione in una variabile che può essere utilizzata in qualche altro punto della query. Questa clausola è utile quando è necessario riutilizzare la stessa espressione più volte nella stessa query e non si vuole definirla ogni singola volta che la si utilizza. Utilizzando la clausola *let*, si può definire una nuova variabile range per quell'espressione e referenziarla nella query. Una volta assegnata, una variabile range definita da una clausola *let* non può essere modificata. Tuttavia, se la variabile range contiene un tipo interrogabile, può essere interrogata. Nel Listato 2-21, si può vedere un esempio di questa clausola applicata per selezionare le stesse categorie di prodotto con il conteggio dei relativi prodotti, ordinati per contatore.

Listato 2-21 Un esempio C# 3.0 di utilizzo della clausola *let*

```
var categoriesByProductsNumberQuery =
    from c in categories
    join p in products on c.IdCategory equals p.IdCategory
    into productsByCategory
    let ProductsCount = productsByCategory.Count()
```

```

    orderby ProductsCount
    select new { c.IdCategory, ProductsCount};

foreach (var item in categoriesByProductsNumberQuery) {
    Console.WriteLine(item);
}

```

Ecco l'output del precedente estratto di codice:

```

{ IdCategory = 3, ProductsCount = 0 }
{ IdCategory = 2, ProductsCount = 2 }
{ IdCategory = 1, ProductsCount = 3 }

```

Visual Basic 2008 utilizza una sintassi molto simile a C# 3.0 e vi permette anche di definire più alias, separati da virgole, nella stessa clausola *let*.

Ulteriori parole chiave Visual Basic 2008

Visual Basic 2008 comprende ulteriori parole chiave per le espressioni query che sono disponibili in C# 3.0 solo utilizzando gli extension method. Queste parole chiave sono descritte nel seguente elenco:

- *Aggregate*, che è utile per applicare una funzione di aggregazione a un datasource. Può essere utilizzata per iniziare una nuova query al posto di una clausola *From*.
- *Distinct*, che può essere utilizzata per eliminare valori duplicati nei risultati della query.
- *Skip*, che può essere utilizzata per saltare i primi *N* elementi del risultato di una query.
- *Skip While*, che può essere utilizzata per saltare i primi elementi del risultato di una query che verificano un predicato fornito.
- *Take*, che può essere utilizzata per estrarre i primi *N* elementi del risultato di una query.
- *Take While*, che può essere utilizzato per estrarre i primi elementi del risultato di una query che verificano un predicato fornito.

Skip e *Take*, o *Skip While* e *Take While*, possono essere utilizzati assieme per paginare i risultati della query. Torneremo su questa questione con alcuni esempi nel Capitolo 3.

Ulteriori informazioni sulla sintassi delle query

A questo punto, avete visto tutte le parole chiavi per le query disponibili attraverso i linguaggi di programmazione. Tuttavia, ricordate che ciascuna espressione query viene convertita dal compilatore del linguaggio in un'invocazione ai corrispondenti extension method. Ogni qualvolta è necessario interrogare un datasource utilizzando LINQ e non vi è alcuna keyword disponibile per una particolare operazione in un'espressione query, si possono utilizzare direttamente metodi di estensione nativi o personalizzati in combinazione con la sintassi dell'espressione query. Se si utilizzano solo gli extension method (come è mostrato nel Listato 2-3), la sintassi è detta *method syntax* (ossia

“sintassi di metodo”). Se si utilizza la sintassi della query in combinazione con gli `extension method` (come è mostrato nel Listato 2-17), il risultato è definito `mixed query syntax` (ossia “sintassi mista di query”).

Valutazione differita della query e risoluzione dei metodi di estensione

In questi paragrafi, vogliamo esaminare due comportamenti di un’espressione query: la valutazione differita della query e la risoluzione dei metodi di estensione. Entrambi questi concetti sono importanti per tutte le implementazioni LINQ.

Valutazione differita della query

Un’espressione query non viene valutata quando viene definita ma quando viene utilizzata. Si consideri l’esempio nel Listato 2-22.

Listato 2-22 Una query LINQ d’esempio su un insieme di sviluppatori

```
List<Developer> developers = new List<Developer>(new Developer[] {
    new Developer { Name = "Paolo", Language = "C#", Age = 32 },
    new Developer { Name = "Marco", Language = "C#", Age = 37},
    new Developer { Name = "Frank", Language = "VB.NET", Age = 48 }
});

var query =
    from d in developers
    where d.Language == "C#"
    select new { d.Name, d.Age };

Console.WriteLine("Ci sono {0} sviluppatori C#.", query.Count());
```

Questo codice dichiara una query molto semplice che contiene solo due elementi, come si può vedere leggendo il codice che dichiara l’elenco degli sviluppatori o semplicemente verificando l’output nella console del codice che invoca il metodo di estensione `Count`.

Ci sono 2 sviluppatori C#.

Si immagina ora di voler modificare il contenuto della sequenza sorgente aggiungendo una nuova istanza `Developer`, dopo che la variabile `query` è stata definita (come è mostrato nel Listato 2-23).

Listato 2-23 Codice d’esempio per modificare l’insieme degli sviluppatori che stiamo interrogando

```
developers.Add(new Developer {
    Name = "Roberto", Language = "C#", Age = 35 });

Console.WriteLine("Ci sono {0} sviluppatori C#.", query.Count());
```

Se enumeriamo nuovamente la variabile *query* o controlliamo semplicemente il conteggio degli elementi, come facciamo nel Listato 2-23 dopo che è stato aggiunto un nuovo sviluppatore, il risultato è tre. Lo sviluppatore che abbiamo aggiunto ora viene incluso nel risultato anche se è stato aggiunto dopo la definizione di *query*.

Il motivo di questo comportamento è che da un punto di vista logico, un'espressione query descrive una sorta di "piano della query". Non viene effettivamente eseguito finché non viene utilizzata la query, e sarà eseguito ripetutamente ogni volta che la si esegue. Alcune implementazioni LINQ, come LINQ to Objects, implementano questo comportamento attraverso delegate. Altre, come LINQ to SQL, possono utilizzare alberi di espressioni che sfruttano l'interfaccia *IQueryable<T>*. Chiamiamo questo comportamento *valutazione differita della query*, ed è un concetto fondamentale in LINQ, indipendentemente dall'implementazione LINQ che si sta utilizzando.

La valutazione differita della query è utile poiché si possono definire una volta le query e applicarle diverse volte: se la sequenza sorgente è stata modificata, il risultato sarà sempre aggiornato al contenuto più recente. Tuttavia, si consideri una situazione in cui si vuole un'istantanea del risultato a un particolare "safe point" da utilizzare molte volte, evitando la riesecuzione per motivi di prestazioni o per essere indipendenti dalle modifiche alla sequenza sorgente. È necessario effettuare una copia del risultato, cosa che si può fare utilizzando un'insieme di operatori, detti operatori di conversione (come *ToArray*, *ToList*, *ToDictionary*, *ToLookup*), che sono specifici per questo scopo. Tratteremo gli operatori di conversione in dettaglio nel Capitolo 3.

Risoluzione dei metodi di estensione

La risoluzione dei metodi di estensione è uno dei concetti più importanti da comprendere se si vuole padroneggiare LINQ. Si consideri il codice nel Listato 2-24, in cui definiamo una lista personalizzata di tipo *Developer* (denominata *Developers*) e una classe, *DevelopersExtension*, che fornisce un metodo di estensione denominato *Where* che si applica in modo specifico alle istanze del tipo *Developers*.

Listato 2-24 Codice d'esempio per modificare l'insieme degli sviluppatori che stiamo interrogando

```
public sealed class Developers : List<Developer> {
    public Developers(IEnumerable<Developer> items) : base(items) { }
}

public static class DevelopersExtension {
    public static IEnumerable<Developer> Where(
        this Developers source, Func<Developer, bool> predicate) {

        Console.WriteLine("Invoked Where extension method for Developers");
        return (source.AsEnumerable().Where(predicate));
    }

    public static IEnumerable<Developer> Where(
        this Developers source,
        Func<Developer, int, bool> predicate) {
```

```

        Console.WriteLine("Invoked Where extension method for Developers");
        return (source.AsEnumerable().Where(predicate));
    }
}

```

Il solo lavoro speciale che svolgiamo negli extension method *Where* personalizzati è scrivere nella console per indicare che sono stati eseguiti. Dopodiché passiamo la richiesta agli extension method *Where* definiti per una istanza standard del tipo *IEnumerable<T>*, convertendo la sorgente con un metodo denominato *AsEnumerable*, che tratteremo nel Capitolo 3.

Se utilizziamo il solito array *developers*, il comportamento della query nel Listato 2-25 è abbastanza interessante.

Listato 2-25 Un'espressione query su un elenco personalizzato di tipo *Developers*

```

Developers developers = new Developers(new Developer[] {
    new Developer { Name = "Paolo", Language = "C#", Age = 32 },
    new Developer { Name = "Marco", Language = "C#", Age = 37},
    new Developer { Name = "Frank", Language = "VB.NET", Age = 48 },
});

var query =
    from d in developers
    where d.Language == "C#"
    select d;

Console.WriteLine("Ci sono {0} sviluppatori C#.", query.Count());

```

L'espressione query verrà convertita dal compilatore nel seguente codice, come abbiamo visto prima in questo capitolo:

```

var query =
    developers
    .Where(d => d.Language == "C#")
    .Select(d => d);

```

Come conseguenza della presenza della classe *DevelopersExtension*, l'extension method *Where* è quello definito da *DevelopersExtension*, invece di quello general-purpose definito in *System.Linq.Enumerable*. (Per essere considerata come una classe container di extension method, la classe *DevelopersExtension* deve essere dichiarata *static* ed essere definita nel namespace corrente o in un namespace incluso nelle direttive *using* attive). Il codice risultante prodotto dal compilatore che risolve gli extension method è il seguente.

```

var query =
    Enumerable.Select(
        DevelopersExtension.Where(
            developers,
            d => d.Language == "C#"),
        d=> d);

```

In definitiva, stiamo sempre invocando metodi static di una classe static, ma la sintassi richiesta è più leggera e più intuitiva con gli extension method che utilizzando

le prolisse invocazioni esplicite a metodi static.

Stiamo ora sperimentando la reale potenza di LINQ. Utilizzando gli extension method, siamo in grado di definire comportamenti custom per specifici tipi. Nei capitoli seguenti, discuteremo di LINQ to SQL, di LINQ to XML, e di altre implementazioni di LINQ. Queste implementazioni sono appunto implementazioni specifiche degli operatori di interrogazione, grazie alla risoluzione degli extension method realizzata dai compilatori.

A questo punto, tutto sembra filare liscio. Ma ora immaginate di dover interrogare la lista custom di tipo *Developers* con il metodo di estensione *Where* standard piuttosto che con quello specializzato. Si dovrebbe convertire la lista custom in una più generalizzata per dirottare la risoluzione dell'extension method effettuata dal compilatore. Si tratta di un ulteriore scenario che può trarre vantaggio dagli operatori di conversione, che tratteremo nel Capitolo 3.

Alcune considerazioni finali sulle query LINQ

In questi paragrafi, tratteremo alcuni ulteriori dettagli sulle espressioni query degeneri e sulla gestione delle eccezioni.

Espressioni query degeneri

Talvolta è necessario iterare sugli elementi di un datasource senza alcun filtraggio, ordinamento, raggruppamento o proiezione personalizzata. Si consideri ad esempio la query presentata nel Listato 2-26.

Listato 2-26 Un'espressione query degenera su una lista di tipo *Developers*

```
Developer[] developers = new Developer[] {
    ...
};

var query =
    from d in developers
    select d;

foreach (var developer in query) {
    Console.WriteLine(developer.Name);
}
```

In questo estratto di codice, iteriamo semplicemente sul datasource, pertanto qualcuno potrebbe chiedersi perché non utilizzare direttamente il datasource, come facciamo nel Listato 2-27.

Listato 2-27 Iterazione su una lista di tipo *Developers*

```
Developer[] developers = new Developer[] {
    ...
};

foreach (var developer in developers) {
    Console.WriteLine(developer.Name);
}
```

Apparentemente, i risultati di entrambi i Listati 2-26 e 2-27 sono gli stessi. Tuttavia, nel Listato 2-26, l'utilizzo dell'espressione query assicura che se esiste uno specifico metodo di estensione *Select* per il datasource, verrà invocato il metodo personalizzato e il risultato sarà consistente in conseguenza alla traduzione dell'espressione query nella corrispondente sintassi di metodo.

Una query che restituisce semplicemente un risultato uguale al datasource originale (apparendo così banale o inutile) è detta *espressione query degenera*. Viceversa, iterando direttamente sul datasource (come nel Listato 2-27) si salta l'invocazione di un extension method *Select* personalizzato e non garantisce il comportamento corretto a meno che non si voglia iterare esplicitamente sul datasource senza utilizzare LINQ.

Gestione delle eccezioni

Le espressioni query possono referenziare metodi esterni nella propria definizione. Prima o poi questi metodi possono fallire. Si consideri la query definita nel Listato 2-28, in cui invociamo il metodo *DoSomething* su ciascun elemento del datasource.

Listato 2-28 Un'espressione query C# 3.0 basata su un metodo esterno che genera un'eccezione fittizia

```
static Boolean DoSomething(Developer dev) {
    if (dev.Age > 40)
        throw new ArgumentOutOfRangeException("dev");
    return (dev.Language == "C#");
}

static void Main() {
    Developer[] developers = new Developer[] {

        new Developer { Name = "Frank", Language = "VB.NET", Age = 48 },
        ...
    };

    var query =
        from d in developers
        let SomethingResult = DoSomething(d)
        select new { d.Name, SomethingResult };

    foreach (var item in query) {
        Console.WriteLine(item);
    }
}
```

Il metodo *DoSomething* genera un'eccezione fittizia per ogni sviluppatore che ha più di 40 anni. Questo metodo lo invociamo dalla query. Durante l'esecuzione della query, quando si giunge all'iterazione sullo sviluppatore Frank, che ha 48 anni, il metodo custom genererà un'eccezione.

Prima di tutto, si dovrà considerare attentamente l'invocazione di metodi personalizzati nelle definizioni delle query poiché è un'abitudine pericolosa, come si può vedere nell'eseguire questo codice d'esempio. Tuttavia, nei casi in cui si decide di invocare metodi esterni, il modo migliore per interagire con questi metodi è di

racchiudere l'enumerazione del risultato della query in un blocco *try ... catch*. Infatti, come avete appena visto nel paragrafo “Valutazione differita della query”, un'espressione query viene eseguita ogni volta che viene enumerata e non quando viene definita. Così, il modo corretto di scrivere il codice nel Listato 2-28 è presentato nel Listato 2-29.

Listato 2-29 Un'espressione query C# 3.0 utilizzata con la gestione delle eccezioni

```
Developer[] developers = new Developer[] {
    new Developer { Name = "Frank", Language = "VB.NET", Age = 48 },
    ...
};

var query =
    from d in developers
    let SomethingResult = DoSomething(d)
    select new { d.Name, SomethingResult };

try {
    foreach (var item in query) {
        Console.WriteLine(item);
    }
}
catch (ArgumentOutOfRangeException e) {
    Console.WriteLine(e.Message);
}
```

In genere, è inutile racchiudere la definizione di un'espressione query con un blocco *try ... catch*. Ancor più, per lo stesso motivo si dovrà evitare di utilizzare direttamente i risultati dei metodi o dei costruttori come datasource per un'espressione query e si dovrà invece assegnare i risultati a variabili di istanza, avvolgendo l'assegnazione della variabile con un blocco *try ... catch* come facciamo nel Listato 2-30.

Listato 2-30 Un'espressione query C# 3.0 con la gestione delle eccezioni nella dichiarazione di variabili locali

```
static void queryWithExceptionHandledInDataSourceDefinition() {
    Developer[] developers = null;

    try {
        developers = createDevelopersDataSource();
    }
    catch (InvalidOperationException e) {
        // Si immagini che createDevelopersDataSource
        // generi una InvalidOperationException in caso di errore

        // La gestisce in qualche modo ...
        Console.WriteLine(e.Message);
    }

    if (developers != null)
    {
        var query =
            from d in developers
            let SomethingResult = DoSomething(d) select new {
```

```
        d.Name, SomethingResult });  
  
    try {  
        foreach (var item in query) {  
            Console.WriteLine(item);  
        }  
    }  
    catch (ArgumentOutOfRangeException e) {  
        Console.WriteLine(e.Message);  
    }  
}  
  
private static Developer[] createDevelopersDataSource() {  
    // generata la InvalidOperationException fittizia  
    throw new InvalidOperationException();  
}
```

Riepilogo

In questo capitolo, abbiamo discusso i principi delle espressioni query e le relative versioni differenti della sintassi (sintassi di query, sintassi di metodo, e sintassi mista), così come le principali parole chiave relative alle query disponibili in C# 3.0 e in Visual Basic 2008. Abbiamo discusso due importanti caratteristiche LINQ: la valutazione differita delle query e la risoluzione dei metodi di estensione. E avete anche visto degli esempi di espressioni query degeneri e come gestire le eccezioni mentre si enumerano le espressioni query. Nel prossimo capitolo, ci rivolgeremo a LINQ to Objects in dettaglio.