16

Caratteristiche specifiche dei linguaggi

Per qualcosa di più esauriente ed esatto visita http://it.wikipedia.org/wiki/Linguaggio_di_programmazione#Bytecode_e_P-code

ABBIAMO QUI

- **Come scegliere il linguaggio** più adatto al lavoro da fare
- Lavorare con il nuovo **linguaggio C #** e caratteristiche del linguaggio **VB**
- Capire e iniziare con il linguaggio **Visual F** #

L'ecosistema per i linguaggi .NET è vivo e vegeto. Con centinaia di linguaggi che si adattano a .NET Framework (potete trovarne una lista abbastanza completa in www.dotnetpowered.com/languagesaspx), gli sviluppatori .NET hanno un enorme arsenale di lingua a loro disposizione. Perché il .NET Framework è stato progettato con l'interoperabilità lingua in mente, queste lingue sono anche in grado di comunicare tra loro, consentendo un impollinazione incrociata creativa di lingue attraverso una sezione trasversale di problemi di programmazione. Si può letteralmente scegliere lo strumento lingua giusta per il lavoro.
Questo capitolo esplora alcune delle più recenti paradigmi linguistici all'interno dell'ecosistema, ognuna con particolari caratteristiche che rendono solo un po' più facile risolvere quei problemi di programmazione duri. Dopo un tour di alcuni dei paradigmi linguaggio di programmazione, imparerete a conoscere alcune delle nuove caratteristiche del linguaggio introdotte in Visual Studio 2012.

Colpire un chiodo con il martello GIUSTO

Bisogna essere un programmatore flessibile e diversificato. Il paesaggio di programmazione richiede eleganza, efficienza e longevità. Sono finiti i giorni quando sceglievi una lingua e una piattaforma e lavoravi come un matto per soddisfare i requisiti del tuo insieme di problemi. Chiodi Diversi a volte richiedono martelli diversi. Dato che sulla piattaforma .NET sono disponibili centinaia di lingue, cosa li rende diverse l'una dall'altra? A dire il vero, la maggior parte sono piccole evoluzioni di lingue precedenti e non sono particolarmente utili in un ambiente aziendale. Tuttavia, è facile classificare le lingue in una gamma di paradigmi di programmazione.

I linguaggi di programmazione possono essere classificate in vari modi, ma adottando un approccio di massima, si può raggruppare i linguaggi in quattro grandi categorie: **imperative, dichiarative, dinamiche e funzionali**. Questa sezione dà un rapido sguardo a queste categorie e alle lingue che vi si adattano.

Categoria imperativa (C, C ++, VB, C #).

Essa comprende le lingue in cui i comandi di linguaggio principalmente manipolano lo stato del programma, linguaggi orientati agli oggetti che sono manipolatori classici focalizzati sulla creazione e la modifica di oggetti**. I linguaggi C e C ++ si adattano perfettamente nella categoria imperativa, come i più diffusi VB e C #.**

I linguaggi imperativi sono progettati per aumentare il livello di astrazione dal codice macchina. Si dice che quando Grace Hopper inventò la prima volta il compilatore, i colleghi della programmazione lamentarono che ciò avrebbe fatto perdere il posto di lavoro.

Sono adatti a descrivere gli scenari del mondo reale attraverso il sistema ad oggetti. Sono rigorosi - ovvero il compilatore fa un sacco di controlli di sicurezza. Ciò significa che non è possibile modificare facilmente un tipo ‘mucca’ a un tipo ‘pecora’ - così, ad esempio, se si dichiara che è necessario il tipo ‘mucca’ nella scrittura di un metodo, il compilatore

controlla che non sia scritto il tipo sbagliato, come anche gli errori di sintassi e di linguaggio. Di solito hanno meccanismi di riutilizzo, pezzi di codice possono essere facilmente individuati in modo che altri percorsi di codice, li possono usare all'interno dello stesso modulo o anche per progetti diversi. Essi sono i più popolari. Sono chiaramente una buona scelta quando per lavorare su un problema c'è bisogno di un team di persone. Tipici linguaggi imperativi sono anche:

APL	Assembly	ALGOL	COBOL
B	BASIC	BCPL	FORTRAN
Forth	Hot soup processor	PL/I	POP

Categoria Dichiarativa (HTML, XAML, SQL in parte,)

Nei linguaggi dichiarativi o logici l'istruzione è una clausola che descrive una relazione fra i dati: programmare in un linguaggio logico significa descrivere l'insieme delle relazioni esistenti fra i dati e il risultato voluto, e il programmatore è impegnato nello stabilire in che modo i dati devono evolvere durante il calcolo. Non c'è un ordine prestabilito di esecuzione delle varie clausole, ma è compito dell'interprete trovare l'ordine giusto. Esempi di linguaggi dichiarativi (logici)

HTML	**SQL**	**XAML**	Curry	Mercury	Prolog

Questi tipi di linguaggio descrivono il '**COSA**' contiene il layout di una pagina, piuttosto che il '**COME**' (che viene descritto dai linguaggi imperativi attraverso dichiarazioni che manipolano lo stato del programma).
Il **'COSA'** è il layout di una pagina che comprende **font, testo, e decorazione,** e dove devono essere mostrate le immagini.
il linguaggio SQL: alcune parti di un altro classico, **il linguaggio SQL**, sono dichiarative – perché descrivono ciò che si vuole da un database relazionale.
XAML è un altro esempio di un linguaggio dichiarativo L' **eXtensible Application Markup Language (XAML),** guida una lunga lista di linguaggi dichiarativi basati su **XML.**
I linguaggi dichiarativi sono particolarmente adattii per la descrizione e la trasformazione dei dati, e come tali, li abbiamo usati per anni per recuperare e manipolare i dati per i nostri linguaggi imperativi.

Categoria Dinamica (REPL, IronPython, IronRuby)

AutoIt	Game Maker	JavaScript	Lingo	PHP	Rexx	thinBasic
Applescript	Hybris	JScript	Lua	Python	Ruby	VBA
ActionScript	HyperTalk	mIRC scripting	Perl	QBasic	Tcl	VBScript

La categoria dinamica include tutte le lingue che presentano caratteristiche "dinamicihe", come **Read Eval Print Loops (REPL),** duck typing (tipizzazione non rigorosa, cioè, se un oggetto si presenta come un'anatra e cammina come un'anatra deve essere un'anatra), e altro ancora.
I linguaggi dinamici ritardano il momento della compilazione perchè possono essere eseguiti prima di compilare. Mentre nel tipico **C #** l'invocazione del metodo **Console.WriteLine()** è staticamente controllato e collegato alla fase di compilazione, un linguaggio dinamico lo può eseguire subito. Se non trova il metodo o il tipo, il linguaggio ha funzioni che danno indicazioni al programmatore per collegare un metodo di errore in modo che il programmatore può prendere questi errori di programmazione e provare qualcosa di diverso.

Altre caratteristiche includono oggetti e classi di oggetti che si estendono, e interfacce in fase di esecuzione (che significa modificare al volo il sistema e il tipo); visibilità dinamica (ad esempio, si può accedere a una variabile definita nell'ambito globale mediante metodi privati o nidificati); e altro ancora.
Metodi di compilazione di questo tipo hanno effetti collaterali interessanti. Se i tipi non hanno bisogno di essere completamente definiti in anticipo (perché il sistema tipo è flessibile), è possibile scrivere codice che utilizza le interfacce severe (come COM, o altri, come assembly .NET, per esempio) e rendere quel codice altamente resistente al fallimento.
Da un linguaggio dinamico, è possibile agganciare il "metodo mancante" nel meccanismo del linguaggio, e quando una particolare interfaccia è cambiata, basta fare un po' di ricerca "a freddo" su tale interfaccia e decidere se è possibile che invocare chiamare qualcos'altro. Ciò significa che è possibile scrivere un codice che incolla insieme di interfacce che possono non essere dipendenti dalla versione del linguaggio.

I linguaggi dinamici sono adatti alla prototipazione rapida. Non dover definire i tipi in anticipo (cosa che dovreste fare subito in C #) consente di concentrarsi sul codice per risolvere i problemi, piuttosto che sui vincoli del tipo di attuazione.
Il linguaggio **REPL** consente di scrivere i prototipi riga per riga e vedere immediatamente come i cambiamenti si riflettono nel programma invece di perdere tempo a fare un ciclo di compilazione (run-debug).
Se siete interessati a guardare ai linguaggi dinamici sulla piattaforma .NET, siete fortunati. Microsoft ha rilasciato **IronPython** (www.codeplex.com/IronPython), che è una **implementazione di Python per .NET Framework**. Il linguaggio Python è un classico esempio di linguaggio dinamico ed è molto popolare nel calcolo scientifico, l'amministrazione dei sistemi, e lo spazio di programmazione in generale.
Se Python non solletica la vostra fantasia, è anche possibile scaricare e provare **IronRuby** (www.ironruby.net/), che è un'implementazione del linguaggio **Ruby per il .NET Framework**. Ruby è un linguaggio dinamico molto popolare nello spazio web, e anche se è ancora relativamente giovane, ha un enorme seguito.

Funzionale

La categoria funzionale si concentra sulle lingue che trattano computazione come funzioni matematiche. Essi si sforzano di evitare la manipolazione dello stato, invece di concentrarsi sui risultati di funzioni come i mattoni per risolvere i problemi. Se abbiamo già fatto qualche calcolo prima, la teoria alla base della programmazione funzionale può apparire familiare. Poiché la programmazione funzionale tipicamente non manipola lo stato, l'area degli effetti collaterali generati in un programma è molto più minore.

Ciò significa che va molto bene per implementare algoritmi paralleli e concorrenti. La risorsa fondamentale dei sistemi destinati a lavorare insieme è i fatto di evitare sovrapposizione "non intenzionali" manipolazioni di stato.
La mancata sincronizzazione del codice di manipolazione dello stato provoca blocchi, arresti del programma, e invarianti interrotti, che ne sono manifestazioni classiche.
Programmazione concorrente e sincronizzazione attraverso thread, memoria condivisa, e blocchi è incredibilmente difficile, quindi perché non evitarla?
Poiché la programmazione funzionale incoraggia il programmatore a scrivere algoritmi privi di stato, il compilatore può ragionare sul parallelismo automatico del codice. Cioè è possibile sfruttare la potenza dei processori multi-core, senza il lavoro pesante di gestione di processi in parallelo (thread), dei passaggi, e della memoria condivisa.
I programmi funzionali sono concisi. Di solito serve meno codice per arrivare a una soluzione che con linguaggio cugino di tipo imperativo. Meno codice significa in genere un minor numero di errori e meno superficie da testare.

I linguaggi funzionali sono basati sul concetto matematico di funzione. In un linguaggio funzionale puro l'assegnazione esplicita risulta addirittura completamente assente e si utilizza soltanto il passaggio dei parametri. Tipicamente in tale modello il controllo del calcolo è gestito dalla ricorsione e dal pattern matching, mentre la struttura dati più diffusa è la lista, una sequenza di elementi. Il più importante esponente di questa categoria è senz'altro il Lisp (LISt Processing).

Clarion	Clean	Curry	Haskell	Lisp	Scala
Scheme	Standard ML	Caml	OCaml	C++11	C#

Queste categorie sono ampie intrinsecamente : Le Lingue possono includere caratteristiche comuni a una o più delle categorie sopra descritte. Le categorie devono essere utilizzate come un modo di mettere in relazione caratteristiche del linguaggio esistenti in loro con i problemi particolari che sono capaci di risolvere.
Linguaggi come **C # e VB.NET** sfruttano caratteristiche dalle loro controparti dinamiche e funzionali.
Il Language Integrated Query (LINQ) è un bell'esempio di un paradigma in prestito. Pensuamo alla seguente query **C # 3.0 LINQ**:

```
var query = from c in clienti
where c.CompanyName == "Microsoft"
select new { c.ID, c.CompanyName };
```

Questo ha alcune caratteristiche prese in prestito. La parola chiave **var** dice "dedurre tipo di **query** specificata", che assomiglia molto a qualcosa da un linguaggio dinamico. La query vera e propria, **from c ... in,** sembra e si comporta come il linguaggio dichiarativo,SQL e **select new { c.ID,** … . crea un nuovo tipo anonimo, ancora una volta qualcosa che sembra abbastanza dinamico. I risultati del codice generato di queste affermazioni sono particolarmente interessanti: sono in realtà non compilate nel classico **IL (**linguaggio intermedio); ma stanno invece in quello che è chiamato una espressione ad albero e poi interpretate in fase di esecuzione - qualcosa preso fin dal primo playbook di linguaggio dinamico.

La verità è che queste categorie non sono particolarmente importanti per decidere quale strumento utilizzare per risolvere il problema giusto. In questo momento è di moda l'impollinazione incrociata di set di funzionalità da ogni categoria di linguaggio. Questo è buono per un programmatore, il cui linguaggio favorito in genere raccoglie le migliori caratteristiche da ogni categoria.
Attualmente la tendenza è per i linguaggi imperativo/dinamici che sono destinati a essere utilizzati dagli sviluppatori di applicazioni, mentre i linguaggi funzionali sono applicati nella soluzione di problemi specifici per i domini.
Se sei un programmatore .NET, devi sorridere ancora di più.
L'interoperabilità della Lingua attraverso **Common Language Specification (CLS)** funziona perfettamente, significa che è possibile utilizzare il linguaggio imperativo favorito per la maggior parte dei problemi che stai cercando di risolvere e quindi chiamare in un linguaggio funzionale alla tua manipolazione dei dati, o forse un po 'di hardware di calcolo di base è necessario per risolvere un problema.

DUE LINGUAGGI UNA STORIA

Dalla creazione del .NET Framework, vi è stato un dibattito in corso sul linguaggio che gli sviluppatori dovrebbero usare per scrivere le loro applicazioni.
In molti casi, i gruppi di lavoro scelgono tra C # e VB basadosi sulla conoscenza preventiva di una C / C ++, Java, o VB6. Tuttavia, questa decisione è stata resa più difficile dalla passata divergenza tra i due linguaggi.
In passato, i gruppi di lavoro di Microsoft sui linguaggi facevano aggiunte aile loro lingue indipendentemente, da cui risultava un certo numero di caratteristiche dii un linguaggio assenti in un altro.

Ad esempio, **VB** ha integrato il supporto di lingua per lavorare con **XML** letterali, mentre **C #** ha iteratori e metodi anonimi. Anche se queste caratteristiche aiutavano gli utenti di tali lingue, hanno reso difficile di scegliere la lingua da utilizzare per le organizzazioni.
Infatti, in alcuni casi le organizzazioni hanno smesso di utilizzare un mix di linguaggi per trivare e usare la lingua migliore a seconda del lavoro
Sfortunatamente, questo significa che sia il team di sviluppo ha bisogno di leggere e scrivere entrambe le lingue, o di avere una squadra che lavora con C # e alcune persone che lavorano sul codice VB.

Con Visual Studio 2010 e .NET Framework 4.0, all'interno di Microsoft è stata presa la decisione di far evolvere insieme due linguaggi primari.NET, **cioè C # e VB**. Questa co-evoluzione cerca di minimizzare le differenze di capacita tra i due linguaggi (spesso questo è definito come parità di funzione). Tuttavia non è un tentativo di fondere le due lingue; in realtà, è tutto il contrario. Infatti Microsoft ha chiaramente indicato che ogni lingua può implementare una funzione in un modo diverso per assicurarsi che sia in linea con il modo di scrivere interagire con il linguaggio che gli sviluppatori usano già.
Nei prossimi paragrafi, imparerete a conoscere le caratteristiche del linguaggio che sono state aggiunte in Visual Studio 2012.
Inizierete cercando le caratteristiche comuni a entrambe le lingue prima di passare attraverso modifiche ai linguaggi individuali, la maggior parte delle quali discusse nel contesto della funzione di parità, e in che modo una funzionalità introdotta corrisponde a una funzione già presente nell'altra lingua.

La parola chiave 'Async'

Come già accennato, la scrittura di codice che supporta più thread – operazioni contemporanee - è difficile da realizzare. Almeno, è difficile farlo senza introdurre difetti che può essere difficile da identificare e rimuovere.
Per ultime versioni di **C #,** Microsoft ha lavorato con l'obiettivo di rendere più facile la scrittura di applicazioni multithread. Questo si vede con l'introduzione di classi come **Background Work**er e l'uso diffuso del modello asincrono basato su eventi (Event-based Asynchronous Pattern).. Ciascuno di questi è concentrata sull'idea di eliminare la necessità dello sviluppatore di creare thread come parte del suo codice.
In .**NET 4.0**, **Task Parallel Library (TPL)** ha realizzato alcuni concetti multithreading (come ad esempio la separazione in thread paralleli delle iterazioni del ciclo o query LINQ) rendendoli più facilmente disponibili per lo sviluppatore medio.
Con.**NET 4.5** si fa un passo avanti, con l'introduzione della parola chiave async.
L'obiettivo principale della funzione **Async** è quello di chiamare i metodi in modo asincrono senza bisogno di scriverli di nuovo e senza la necessità di dividere il codice in metodi diversi. Non è che questo lavoro non sia finito. Solo che non lo devi scrivere perché lo sviluppatore si prende cura di questo per te.
In realtà ci sono due parole chiave aggiunte come parte della funzione **Async**. Il modificatore async sulla firma del metodo indica che un particolare metodo restituisce un oggetto o attività o un oggetto Task generico.
La differenza tra i due è che task restituisce void (o niente) e Task generico (in forma di Task <TResult>) restituisce un oggetto di tipo TResult. Questo oggetto rappresenta lo stato di esecuzione del metodo. Come tale, esso contiene informazioni sullo stato del Task. L'idea è che il chiamante può quindi utilizzare queste informazioni per operare su e con le attività in esecuzione.
La seconda parola chiave è **Await.** Questa parola è in realtà un operatore e opera su un Task. Quando questo è terminato, l'esecuzione del metodo corrente viene sospesa finché il metodo asincrono rappresentato nell'operazione npn è terminata. Durante l'attesa, il controllo viene restituito al chiamante del metodo che è sospeso.
Per vedere un esempio in azione, si consideri il seguente metodo denominato **GetContentsAtUrl**., che vuole un URL come parametro e restituisce **un array di byte di contenuti** trovati sul luogo indicato:

C#

```
private byte[] GetContentsAtUrl(string url)
{
        var contents = new MemoryStream();
        var webReq = (HttpWebRequest)WebRequest.Create(url);
        using (var webResp = webReq.GetResponse())
                {
                        using (Stream responseStream = webResp.GetResponseStream())
                                {
                                responseStream.CopyTo(contents);
                                }
                }

return contents.ToArray();

}
```

Questo qui sopra è un metodo sincrono. Pertanto, è necessario attendere la restituzione di una la risposta (avviata dalla chiamata al metodo **GetResponse**) prima che il metodo possa essere completato. E mentre siete in attesa, le operazioni del chiamante sono sospese. Per farlo diventare asincrono bisogna attribuirgli la caratteristica di metodo asincrono con questo codice:

C#

```
private async Task<byte[]> GetContentsAtUrlAsync(string url)
{
        var contents = new MemoryStream();
        var webReq = (HttpWebRequest)WebRequest.Create(url);
        using (WebResponse response = await webReq.GetResponseAsync())
                {
                using (Stream responseStream = response.GetResponseStream())
                        {
                        await responseStream.CopyToAsync(contents);
                        }
                }
return contents.ToArray();
}
```

Primo cambiamento (oltre la parola **async**) è **Task**<byte[]> il valore di ritorno per il metodo. Invece di un array di byte, restituisce un oggetto **Task generico dichiarata con l'array di byte**. Questo gli permette di essere utilizzato come parte dell'operatore **await**.

Secondo cambiamento *GetContentsAtUrlAsync(string url) cioè* il nome del metodo. Si tratta di una convenzione (aggiunta della parola **Async** al nome del metodo), che ha lo scopo di aiutare gli sviluppatori a riconoscere che un metodo può essere chiamato in modo asincrono.

Terzo cambiamento. Chiamata a *GetResponseAsync.*invece che a GetResponse, Questo metodo coinvolge la funzione *GetResponse* in un'attività. GetResponseAsync può essere utilizzata con la parola chiave **await** perché restituisce un TASK, Quando viene eseguita questa istruzione, viene creato un thread separato per eseguire la funzione GetResponse il metodo *GetContentsAtUrlAsync* è sospeso, e il controllo viene restituito all'applicazione chiamante.
Quando il metodo GetResponseAsync è completo, sarà ripreso (il termine corretto è in realtà "continuato") il metodo GetContentsAtUrlAsync, perchè l'esecuzione del programma continua con l'istruzione immediatamente seguente

Quarto cambiamento *CopyToAsync(contents);*invece di *CopyTo(contents);*

Solo perché sia chiaro, i metodi asincroni non bloccano il thread corrente. Questo può sembrare un po' strano, ma ciò che accade nel processo di compilazione è che il resto del metodo (cioè dopo la chiamata await) è costruito come proseguimento. Completata la chiamata del metodo (GetResponseAsync, in questo caso) l'esecuzione continua sul thread originale. Questo elimina anche la necessità di mobilitare i callback sul thread dell'interfaccia utente, come sarebbe stato fatto nella programmazione asincrona di versioni precedenti.

Informazioni del chiamante

Alle volte per trovare informazioni su chi sta chiamando potrebbe essere utile un metodo particolare. In NET 4,5, questo è disponibile attraverso l'uso dell'attributo 'Caller Info'. Si veda il seguente metodo:

C#

```
public void TraceMessage(string message)
        {
        Trace.WriteLine("Message: " + message);
        }
```

In questo caso, un messaggio viene passato e scritto da qualche ascoltatore di traccia. Ma cosa succede se si desidera conoscere il nome del metodo che effettua la chiamata? In .NET 4.5, si dovrebbero aggiungere alcuni parametri per il metodo e decorarli con l'attributo 'Caller Info' come mostrato qui:

C#

```
public void TraceMessage(string message,
        [CallerMemberName] string memberName = "", [CallerFilePath] string sourceFilePath = "", [CallerLineNumber] int
        sourceLineNumber = 0)
{
        Trace.WriteLine("Message: " + message);
        Trace.WriteLine("Member Name: " + memberName);
        Trace.WriteLine("Source File Path: " + sourceFilePath);
        Trace.WriteLine("Source Line Number: " + sourceLineNumber);
}
```

Sono stati aggiunti ai metodi un certo numero di parametri, che in realtà sono parametri opzionali, in quanto se non vengono forniti, vengono dati i valori di default.
CallerMemberName, CallerFilePath, e CallerLineNumber, sono in realtà il nome del metodo, il percorso per il codice sorgente, e il numero di riga all'interno del codice sorgente. Questi valori sono ora disponibili per l'uso desiderato..

VISUAL BASIC

Nello spirito di funzione di parità, due delle nuove caratteristiche offerte di questa versione di Visual Basic sono identichei a quelle in C #. **Caller Info** e **Async** sono inclusi entrambi. La differenza tra il codice **C #** e il codice **VB** è solo sintattica.

C'è una parola chiave **Async** che modifica la dichiarazione del method e un operatore **await** su un oggetto **Task**. E ci sono gli attributi **CallerMemberName, CallerFilePath,** e **CallerLineNumber** che possono essere utilizzati per fornire i valori di parametri opzionali.del metodo Così concentriamoci sulle caratteristiche nuove solo per Visual Basic.

Gli iteratori

Gli iteratori sono stati in giro in C # fino dal Visual Studio 2005, ma sono stati disponibili in Visual Basic più di recente. Essi sono un concetto utilizzato di rado, ma che, quando serve, è molto utile. In poche parole, la parola chiave **iterator** permette allo sviluppatore di creare una iterazione personalizzata attraverso una collezione.
Inizia con un semplice metodo che restituisce un valore di **IEnumerable** e come per una dichiarazione:in un For Each

VB

```
Sub Main()
        For Each number As Integer In GetNumbers()
        Console.Write(number & ",")
        Next
 End Sub

Private Iterator Function GetNumbers() As System.Collections.IEnumerable
        Yield 9
        Yield 12
        Yield 14
        Yield 16
End Function
```

Nel frammento di codice, c'è il metodo chiamato *GetNumbers* che restituisce il valore *IEnumerable*. Il corpo del metodo è solo un insieme di quattro affermazioni *Yield.* Nella subroutine principale, c'è una dichiarazione *For Each* che gira attraverso ognuno degli elementi restituiti da *GetNumbers.*

NOTA: Il comandi del codice hanno in inglese un significato, la cui traduzione in italiano è la seguente:

Funzione Sub Principale
Per ogni numero intero contenuto nella funzione PrendiNumeri()
Scrivi sulla Consoile(numero ,&",")
avanti il Prossimo
Fine della Sub Principale

Privatamente Iterare la Funzione PrendiNumeri() Come Una Collezione Enumerabile di sistema
Yield 9
Yield 12
Yield 14
Yield 16
Fine della funzione

Nel frammento di codice, abbiamo il metodo chiamato *GetNumbers* che restituisce un valore *IEnumerable.* Il corpo del metodo è solo un insieme di quattro affermazioni Yield. Nella subroutine principale, c'è l'istruzione For Each Number che gira attraverso ciascuno degli elementi restituiti da GetNumbers.
il metodo continua ad eseguire immediatamente dopo l'Yield precedente fino a trovare il seguente. Quindi, nel caso del frammento di codice precedente, l'uscita sarebbe 9, 12, 14, e 16.

La parola chiave Global

È possibile creare una gerarchia annidata di spazi dei nomi che possono impedire di avere accesso ad alcuni dei tipi built-in di dati .NET. Ad esempio, si consideri il seguente frammento di codice:

VB

```
Namespace MyNameSpace
        Namespace System
                Class Sample
                        Function getValue() As System.Double
                        Dim d As System.Double
                        Return d
                        End Function
                End Class
```

```
    End Namespace
End Namespace
```

NOTA: Il comandi del codice hanno in inglese un significato, la cui traduzione in italiano è la seguente:

```
Nomedispazio MyNamespace
Sistema di Nomedispazio
    classe di esempio
        Funzione getValue () Come System.Double
            Dim d Come System.Double
            restituisce d
        Fine della Funzione
    Fine della Classe
Fine di NomeSpazio
```

Il tentative di compilazione del codice fallirebbe perché non esiste una classe chiamata Double
La parola chiave **Global** consente di evitare questo problema. Quando viene utilizzato **Global** con un **Namespace**, si dice al compilatore di avviare la ricerca del tipo di dati a cominciare dal livello principale del namespace. Il seguente frammento di codice risolve il problema:

VB

```
Namespace MyNameSpace
    Namespace System
        Class Sample
            Function getValue() As Global.System.Double
                Dim d As Global.System.Double
                Return d
            End Function
        End Class
    End Namespace
End Namespace
```

Si aggiunge la keyword **Global** permettendo al compilatore di estendere la ricerca e trovare il tipo di dati System.Double

PowerPacks di Visual Basic

Una delle difficoltà riportate dagli sviluppatori VB6 è che fare i Task in .NET richiede molti più passaggi o è più complesso di quanto non fosse in VB6. Per incoraggiare gli sviluppatori di VB6 a passare al .NETFramework, VB ha introdotto il **My namespace**, che fornisce una serie di metodi di scelta rapida delle funzioni usate più di frequente.
Il team VB ha anche rilasciato i PowerPack Visual Basic per le versioni precedenti di Visual Studio che aggiungono una serie di controlli utili e altre classi per aiutare gli sviluppatori di VB.
Da quando esiste Visual Studio 2010, il Power Pack di Visual Basic è fornito con il prodotto. Come si può vedere nella Figura 16-1, una scheda aggiuntiva nella casella degli strumenti contiene una serie di controlli di disegno come la Linea, Ovale e Rettangolo. Questi possono essere usati per generare grafici semplici, come quello sul lato destro della Figura 16-1.

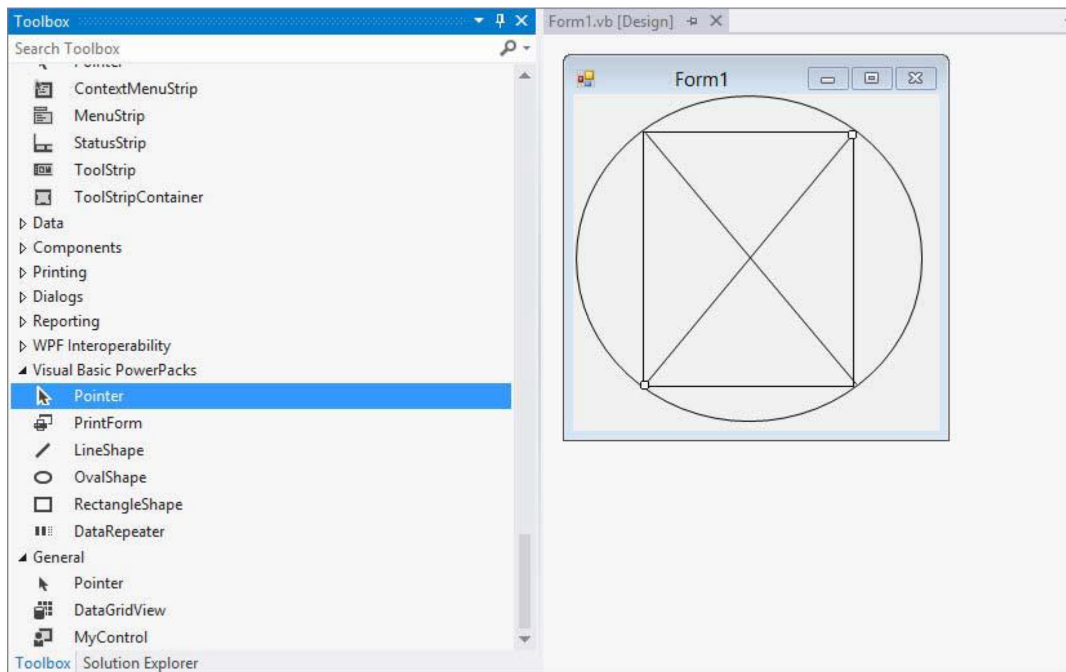


FIGURE 16-1 – generatore di spazi grafici

Anche se i Powerpack per Visual Basic sono disponibili come impostazione predefinita per gli sviluppatori VB, non vi è alcun motivo per cui gli sviluppatori C # non possano accedere agli stessi controlli. Per utilizzare questi controlli in un progetto C #, è sufficiente aggiungere un riferimento all'assembly PowerPack e quindi aggiungere i controlli alla Toolbox. Da lì possono essere usati su qualsiasi applicazione Windows Form.

F#

F # (pronunciato F Sharp) è un linguaggio incubato da Microsoft Research di Cambridge, in Inghilterra, da Don Syme, un ragazzo che ha lavorato su .NET Framework.

F # fornito con Visual Studio 2012 ed è un linguaggio funzionale mulltiparadigma. Ciò significa che è principalmente un linguaggio funzionale, ma supporta anche altri stili di programmazione, come ad esempio stili di **programmazione imperativa e object-oriented** (orientata agli oggetti).

Il vostro primo programma in F#

Accendete Visual Studio 2012 e create un nuovo progetto **F #**. Come mostra la Figura 16-2, il modello **F# Application** si trova nel nodo visivo nella nella finestra di dialogo Nuovo progetto F #. Dategli un nome e fate clic su OK.

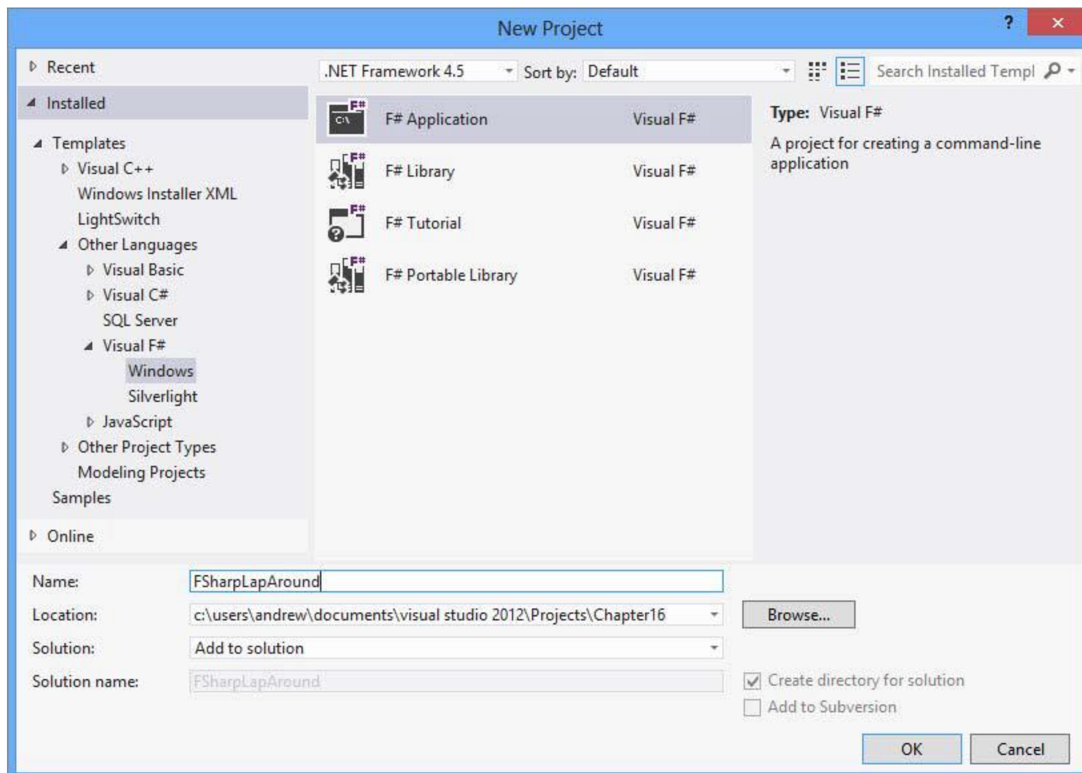


Figura 16-2 - creare un nuovo progetto F #.

Il modello **F # Application** crea semplicemente un progetto **F #** con un singolo file sorgente, **Program.fs**, che è vuoto ad eccezione di un riferimento al F # Developer Center, http://fsharp.net. Se volete saperne di più su F #, un ottimo punto di partenza è il modello F # Tutorial.

Questo crea un normale progetto **F#** tranne che per il file sorgente principale, **Tutorial.fs**, che contiene circa 280 linee di documentazione su come iniziare con F#. Un esercizio in sé interessante consiste nel percorrere questo file e verificare quali caratteristiche del linguaggio e servizi sono disponibili.

Per ora, torniamo al **Program.fs** per ottenere rapidamente il "Ciao Mondo" come esempio installato e funzionante per vedere le varie opzioni disponibili per la compilazione e l'interattività. Aggiungere il seguente codice:

```
#light
printfn "Hello, F# World!"
let x = System.Console.ReadLine();
```

La prima istruzione **, #light**, è un segnale che indica al compilatore che il codice è scritto utilizzando la **sintassi opzionale leggera**. Con questa sintassi, si rende significativa l'indentazione degli spazi, riducendo la necessità di alcuni segni di interruzionei, come as e ;

La seconda dichiarazione scrive semplicemente "Ciao, F # World!" sulla console

> NOTA Se avete lavorato con versioni precedenti di F #, è possibile che il codice ora dia errori di compilazione. F # è nata da un progetto di ricerca e ora è stata trasformata in un'offerta commerciale. Come tale, c'è stato un refactoring del linguaggio, e alcune operazioni sono state spostate fuori da FSharp.Core in strutture di appoggio.
> Ad esempio, il comando print_endline è stato spostato nella Assembly **FSharp.PowerPack.dll**.
> **La F # Powerpack** è disponibile per il download tramite l'F # Developer Center a http://fsharp.net.

Potete avviare un programma F# in due modi. La prima è di eseguire semplicemente l'applicazione come si farebbe normalmente. (Premere F5 per avviare il debug.) Questa compila ed esegue il programma, come mostrato nella Figura 16-3.

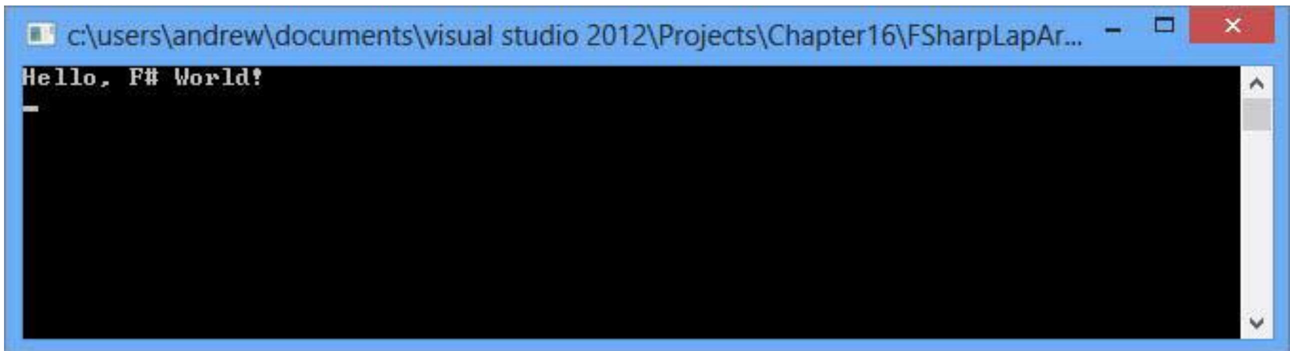


Figura 16-3.

L'altro modo per eseguire un programma di F# è l'utilizzo della finestra di F# Interattiva dall'interno di VisualStudio. Questo permette di evidenziare ed eseguire il codice e vedere immediatamente il risultato nel vostro programma in esecuzione. Consente inoltre di modificare il vostro programma in esecuzione al volo!
La finestra di F# Interattiva (vedi Figura16-4) è disponibile dalla voce di menu **View➪ OtherWindows➪F # Interactive**, o premendo la combinazione di tasti **Ctrl + Alt + F.**
Nella finestra interattiva, è possibile iniziare a interagire con il compilatore F# attraverso il prompt **REPL**. Ciò significa che per ogni linea di F# si digita, si compila ed esegue immediatamente quella linea. I comandi REPLs sono grandi se si vuole testare le idee in modo rapido e modificare i programmi al volo.
Essi consentono una prototipazione rapida e una rapida sperimentazione di algoritmi..

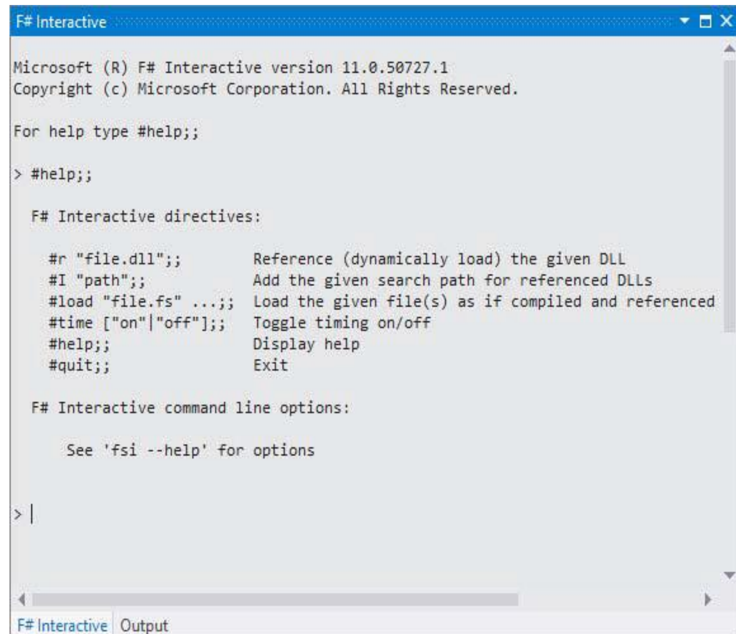


Figura 16-4 – finestra interattiva di F#

Tuttavia, dal prompt **REPL** nella finestra F # Interactive, si perde il valore che Visual Studio fornisce attraverso IntelliSense, frammenti di codice, e così via. La migliore esperienza è quella di entrambi i mondi: utilizzare l'editor di testo di Visual Studio per creare i programmi, e iniettare l'output attraverso il prompt interattivo. È possibile farlo premendo Alt + Invio su qualsiasi pezzo di F # codice sorgente selezionato.

In alternativa, è possibile utilizzare il menu contestuale del tasto destro per inviare una selezione alla finestra interattiva, come mostrato nella Figura 16-5.
Premere Alt + Invio, oppure selezionando Esegui in Interactive, si inietta il codice sorgente evidenziato nella finestra del prompt Interactive e lo esegue immediatamente, come mostrato nella Figura 16-6.
La Figura 16-6 mostra anche il menu contestuale destro del mouse per la finestra F# Interactive dove potete Annullare valutazione interattiva (per le operazioni di lunga durata) o fare il Reset della sessione interattiva (dove verrà scartato qualsiasi stato precedente).

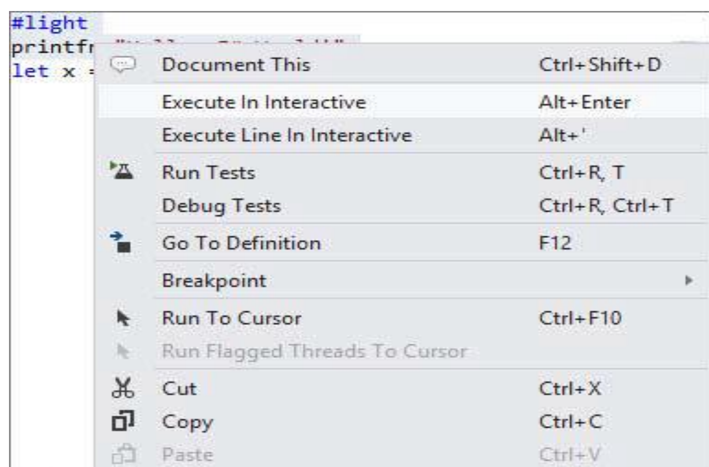
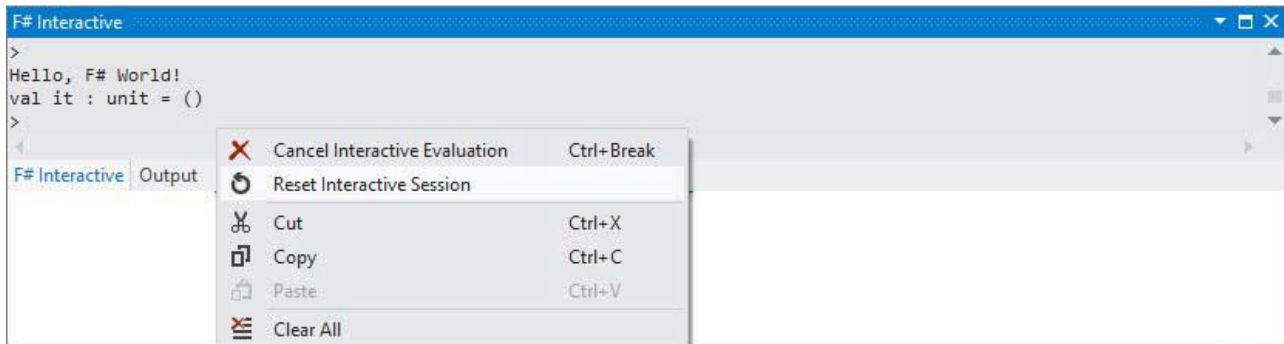


Figura 16-5 – finestra interattiva di F#

Figura 16-6-menu contestuale destro del mouse per la finestra F# Interactive



Explorazione delle caratteristiche del linguaggio F#

Approfondire le nozioni sul linguaggio va oltre lo scopo di questo libro, ma vale la pena di esplorare alcune delle caratteristiche del linguaggio. Semmai, si dovrebbe stuzzicare l'appetito per F# per saperne di più su questa grande lingua.
Un tipo di dati comune nel mondo # F è la lista. Si tratta di un tipo semplice di raccolta con operatori di espressione.
È possibile definire un elenco vuoto, liste multidimensionali, e la classica lista piatta. La lista F# non è modificabile dopo che è stato creata; si può prendere solo una copia.
F# espone una funzione **chiamata di lista** per rendere più facile la creazione, la manipolazione, e rendere le liste e più espressive. Si consideri il seguente:

```
#light
let countInFives = [ for x in 1 .. 20 do if x % 5 = 0 then yield x ]
printf "%A" countInFives
System.Console.ReadLine()
```

L'espressione in parentesi fa un classico loop su una lista che contiene gli elementi da 1 a 20 (il ".." L'espressione **[for x in 1 .. 20 do if x % 5 = 0 then yield x]** è una scorciatoia per la creazione di un nuovo elenco di elementi da 1 a 20 - l'espressione ".." indica l'intervallo di numeri.
L'elenco viene eseguito (**do)** con il ciclo **for** per ogni elemento della lista e il loop termina quando il modulo di x 5 equals 0" qui non capisco cosa vuol dire? In tal modo si definisce un nuovo elenco in una riga

La caratteristica Pattern Matching di F# è un modo flessibile e potente per creare un flusso di controllo. Nel mondo C #, si ha l'interruttore (o semplicemente un gruppo di nidificato di "if else"), ma è di solito vincolato al tipo di quello che si sta trattando. In F# Pattern Matching è simile, ma più flessibile, consentendo ai dati di essere più di qualunque tipo o valori specificati. Ad esempio, vediamo come definire una funzione Fibonacci in F# utilizzando pattern matching:

```
let rec fibonacci x =
match x with
| 0 | 1 -> x
| _ -> fibonacci (x - 1) + fibonacci (x - 2)
printfn "fibonacci 15 = %i" (fibonacci 15)
```

L'operatore pipe (**|**) specifica che si desidera far corrispondere un'espressione sul lato destro del tubo e l'ingresso alla funzione.
La prima espressione dice di restituire l'ingresso della funzione x quando x ha il valore 0 o il valore 1.
La seconda riga dice di restituire il risultato di una chiamata ricorsiva a Fibonacci con un ingresso di x - 1, aggiungendo il numero a un'altra chiamata ricorsiva dove l'input è x - 2.
L'ultima riga scrive il risultato della funzione Fibonacci sulla console.

Successione di Fibonacci: una successione in cui i primi due elementi sono 1 e ogni altro elemento successivo è dato dalla somma dei due che lo precedono

Problema: «Un tale mise una coppia di conigli in un luogo completamente circondato da un muro, per scoprire quante coppie di conigli discendessero da questa in un anno: per natura le coppie di conigli generano ogni mese un'altra coppia e cominciano a procreare a partire dal secondo mese dalla nascita.» Fibonacci, vinse la gara dando al test una risposta così rapida da far persino sospettare che il torneo fosse truccato.
Soluzione
Il primo mese c'è solo una coppia di conigli, il secondo mese ce ne sono 2 di cui una fertile, quindi il terzo ce ne sono 3 di cui 2 fertili, quindi il quarto mese ce ne sono 5 di cui 3 fertili, quindi il quinto mese ce ne sono 8 di cui 5 fertili e così via.
Nasce così la celebre successione di Fibonacci:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Nelle funzioni Pattern matching ha un effetto collaterale interessante - rende l'invio e il controllo del flusso molto più facile e più pulito su diversi tipi di parametri di ricezione. Nel mondo C # / VB.NET si dovrebbe tradizionalmente scrivere una serie di sovraccarichi sulla base di tipi di parametri, ma in F# questo non è necessario in quanto la sintassi di pattern matching consente di ottenere la stessa cosa con una singola funzione.

La **Valutazione pigra** è un'altra caratteristica pulita del linguaggio, comune ai linguaggi funzionali, che anche F# possiede. Significa semplicemente che il compilatore può pianificare la valutazione di una funzione o un'espressione solo quando è necessario, piuttosto che precalcolarla prima per passarla alla funzione. Questo significa che è necessario eseguire solo il codice che è assolutamente necessario - meno cicli di esecuzione e meno lavoro significa più velocità.

In genere, quando si ha un'espressione assegnata a una variabile, l'espressione viene immediatamente eseguita per memorizzare il risultato nella variabile. Sfruttando la teoria che la programmazione funzionale ha effetti collaterali, non vi è alcuna necessità di esprimere immediatamente questo risultato (perché in ordine di esecuzione non è necessario), e come risultato, si deve eseguire solo quando il risultato della variabile è effettivamente necessario. Date un'occhiata a un semplice caso:

```
let lazyDiv = lazy ( 10 / 2 )
printfn "%A" lazyDiv
```

In primo luogo, la parola chiave **Lazy** viene utilizzata per esprimere una funzione o un'espressione che verrà eseguita solo se si è costretti. La seconda linea stampa alla console tutto ciò che c'è in l**azyDiv**.
Se si esegue questo esempio, quello che in realtà come l'uscita della console è "(unevalued)." Questo è perché bisogna forzare, o invocare, l'espressione per ottenere un risultato restituito, come nel seguente esempio:

```
let lazyDiv2 = lazy ( 10 / 2 )
let result = lazyDiv2.Force()
print_any result
```

La funzione *lazyDiv2.Force()* obbliga a eseguire l'espressione *lazyDiv2*. Questo concetto è potente per l'ottimizzazione delle prestazioni delle applicazioni. La riduzione della quantità di lavoro, o di memoria che comporta costruire un'applicazione è estremamente importante nel miglioramento sia delle prestazioni di avvio che di quelle di funzionamento.
Lazy evaluation è anche un concetto necessario quando si tratta di grandi quantità di dati. Per scorrere terabyte di dati memorizzati su disco, una valutazione Lazy esamina i dati in modo da prenderli per fare il backup i solo quando effettivamente serve.
Il Gruppo Applicazioni per Giochi di Microsoft Research ha un grande write-up che usa funzionalità di valutazione pigra in F# con esattamente questo scenario:
http://blogs.technet.com/apg/archive/2006/11/04/dealing-with-terabyteswith-f.aspx

Provveditore di Tipo

La nozione di provveditore di tipo, applicata a F#, è relativamente semplice. Lo sviluppo moderno comporta portare dati da un certo numero di fonti disparate. E per lavorare questi dati, devono essere raggruppati in classi e oggetti che possono essere manipolati dall'applicazione.
Creare manualmente tutte queste classi non è solo noioso, ma aumenta anche la possibilità di errori. Spesso, un generatore di codice potrebbe essere utilizzato per risolvere questo problema. Ma se si utilizza F# in modalità interattiva, i generatori di codice tradizionali non sono i migliori. Ogni volta che si regola un riferimento al servizio il codice avrebbe bisogno di essere rigenerato, il che può essere fastidioso.
Per far fronte a questo, F# ha introdotto una serie di provveditori di tipo vòlti ad affrontare le situazioni di accesso ai dati più comuni. Questi includono l'accesso a database relazionale SQL, ai servizi Open Data (OData), e servizi definiti WSDL. Si ha anche la possibilità di creare e utilizzare il proprio provider di tipo personalizzato.
Come esempio di utilizzo dei fornitori di tipo per accedere a un database di SQL Server, si consideri il seguente codice:

F#

```
#r "System.Data.dll"
#r "FSharp.Data.TypeProviders.dll"
#r "System.Data.Linq.dll"
type dbSchema = SqlDataConnection<"Data Source=.\SQLEXPRESS;Initial
Catalog=AdventureWorksLT2008R2;Integrated Security=SSPI;">
let db = dbSchema.GetDataContext()
let qry =
query {
for row in db.Customers do
select row
}
qry |> Seq.iter (fun row -> printfn "%s, %s" row.Name row.City)
;;
```

NOTA *Per eseguire il codice di cui sopra, è necessario aggiungere una serie di riferimenti al progetto. Questo è vero anche se lo si esegue in modalità F # Interactive. In particolare, devono essere aggiunti i pacchetti System.Data e System.Data.Linq.Per la classe SQLDataConnection e relativa funzionalità dei dati F #, è necessario aggiungere il pacchetto FSharp.Data.TypeProviders*

Dopo aver aggiunto i riferimenti necessari al fornitore di tipo, ai namespace si accede attraverso l'uso della dichiarazione di tipo. Questo permette da creare come tipo, la variabile **dbschema** che contiene tutti il tipi generati che rappresentano le tabelle del database nel database **AdventureWorksLT2008 R2**.
E dopo richiamato **GetDataContext**, la variabile db ha **come proprietà tutti i nomi delle tabelle**, permettendo per ciascuna di esse di esaminare tutte le righe da stampare per nome dei clienti. e per città

Espressioni di Query

Alle versioni precedenti di F # mancava il supporto per LINQ. Come molti sviluppatori C # e VB hanno scoperto, LINQ è una sintassi utile per l'interrogazione di molte fonti di dati diverse e per modellare i dati risultanti come richiesto dall'applicazione. Con LINQ possono essere costruite ed eseguite le Query F# 3.0, ampliando l'espressività del linguaggio. Si consideri il seguente frammento di codice:

F#

```
open System
open System.Data
open System.Data.Linq
open Microsoft.FSharp.Data.TypeProviders
open Microsoft.FSharp.Linq
type dbSchema = SqlDataConnection<"Data Source=.\SQLEXPRESS;Initial
Catalog=AdventureWorksLT2008R2;Integrated Security=SSPI;">
let db = dbSchema.GetDataContext()
let qry =
query {
for row in db.Customers do
where (row.City == "London")
select row
}
qry |> Seq.iter (fun row -> printfn "%s, %s" row.Name row.City)
```

Questo frammento di codice esegue più o meno la stessa funzione di quello riscontrato nella sezione precedente. La differenza è che vengono visualizzati solo i clienti della città di Londra.
Ma la sintassi LINQ è visibile nella dichiarazione di query, tra cui la possibilità di filtrare le righe della tabella in base a criteri specifici. E anche se le parole chiave sono un po' diverse, è disponibile anche la maggior parte delle stesse funzionalità LINQ che si trovano anche in C # e VB.

Proprietà Auto-Implementate

Le proprietà in F # possono essere definite in due modi. La differenza è se si desidera o meno che la proprietà abbia un proprio archivio di backup esplicito. Il modo "tradizionale" per creare una proprietà è la definizione di una variabile privata che contiene il valore della proprietà.
Allora questo valore può essere estratto attraverso il metodo GET e della proprietà impostata. Se, d'altra parte, non è necessario o non si desidera creare quella variabile privata, F # in grado di generarne una. Questo è il concetto che sta dietro le proprietà di auto-implementazione. Il seguente frammento di codice mostra sia l'approccio tradizionale che quello auto-implementato:

F#

```
type Person() =
member val FirstName
with get () = privateFirstName
and set (value) = privateFirstNam <- value
member val LastName = "" with get, set
```

L'ultima riga ha la proprietà di essere auto-implementa. A differenza della proprietà **FirstName**, che ha **get** esplicito e metodi stabiliti e usa la variabile **privateFirstName** come archivio di backup LastName è definita come riferita a una stringa vuota e usa qualsiasi variabile che il compilatore genera automaticamente.

SUMMARY

In questo capitolo si è appreso qualcosa sui diversi stili di linguaggi di programmazione e sui loro punti di forza e di debolezza. Visual Studio 2012 riunisce i due linguaggi .NET primari C # e VB, con l'obiettivo di raggiungere la parità di funzionalità.
La co-evoluzione di queste lingue può aiutare a ridurre i costi di team e dei progetti di sviluppo, permettendo agli sviluppatori di passare più facilmente da una lingua all'altra.
Si è anche imparato su Visual F # che se aumenta la difficoltà dei problemi che si cerca di risolvere aumenta anche la complessità introdotta dalla necessità di scrivere applicazioni altamente parallele. È possibile utilizzare Visual F# per affrontare questi problemi attraverso l'esecuzione di operazioni parallele senza aumentare la complessità di un'applicazione.