EBook Gratuito

APPRENDIMENTO

C# Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#c#

Sommario

Di

Capitolo 1: Iniziare con C # Language

Osservazioni

C # è un linguaggio di programmazione discendente multi-paradigma di Microsoft. C # è un linguaggio gestito che compila in CIL , un bytecode intermedio che può essere eseguito su Windows, Mac OS X e Linux.

Le versioni 1.0, 2.0 e 5.0 sono state standardizzate da ECMA (come ECMA-334) e gli sforzi di standardizzazione per il moderno C # sono in corso.

Versioni

Versione	Data di rilascio
1.0	2002-01-01
1.2	2003-04-01
2.0	2005-09-01
3.0	2007-08-01
4.0	2010-04-01
5.0	2013/06/01
6.0	2015/07/01
7.0	2017/03/07

Examples

Creazione di una nuova applicazione console (Visual Studio)

1. Apri Visual Studio
2. Nella barra degli strumenti, vai su **File** → **Nuovo progetto**
3. Seleziona il tipo di progetto **Applicazione console**
4. Aprire il file `Program.cs` in Solution Explorer
5. Aggiungi il seguente codice a `Main()` :

```
public class Program
{
    public static void Main()
    {
        // Prints a message to the console.
```

```
        System.Console.WriteLine("Hello, World!");

        /* Wait for the user to press a key. This is a common
           way to prevent the console window from terminating
           and disappearing before the programmer can see the contents
           of the window, when the application is run via Start from within VS. */
        System.Console.ReadKey();
    }
}
```

6. Nella barra degli strumenti, fare clic su **Debug** -> **Avvia debug** o premere **F5** o **ctrl + F5** (in esecuzione senza debugger) per eseguire il programma.

Demo dal vivo su ideone

Spiegazione

- `class Program` è una dichiarazione di classe. Il `Program` classe contiene i dati e le definizioni dei metodi utilizzati dal programma. Le classi generalmente contengono più metodi. I metodi definiscono il comportamento della classe. Tuttavia, la classe `Program` ha solo un metodo: `Main` .

- `static void Main()` definisce il metodo `Main` , che è il punto di ingresso per tutti i programmi C #. Il metodo `Main` indica cosa fa la classe quando viene eseguita. È consentito un solo metodo `Main` per classe.

- `System.Console.WriteLine("Hello, world!");` metodo stampa un dato dato (in questo esempio, `Hello, world!`) come output nella finestra della console.

- `System.Console.ReadKey()` , assicura che il programma non si chiuda immediatamente dopo la visualizzazione del messaggio. Lo fa aspettando che l'utente preme un tasto sulla tastiera. Qualsiasi tasto premuto dall'utente interromperà il programma. Il programma termina quando ha terminato l'ultima riga di codice nel metodo `main()` .

Usando la riga di comando

Per compilare tramite la riga di comando utilizzare `MSBuild` o `csc.exe` *(il compilatore C #)* , entrambe le parti del pacchetto Microsoft Build Tools .

Per compilare questo esempio, eseguire il seguente comando nella stessa directory in cui si trova `HelloWorld.cs` :

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs
```

Può anche essere possibile che tu abbia due metodi principali all'interno di un'applicazione. In questo caso, si deve dire al compilatore quale metodo principale per eseguire digitando il

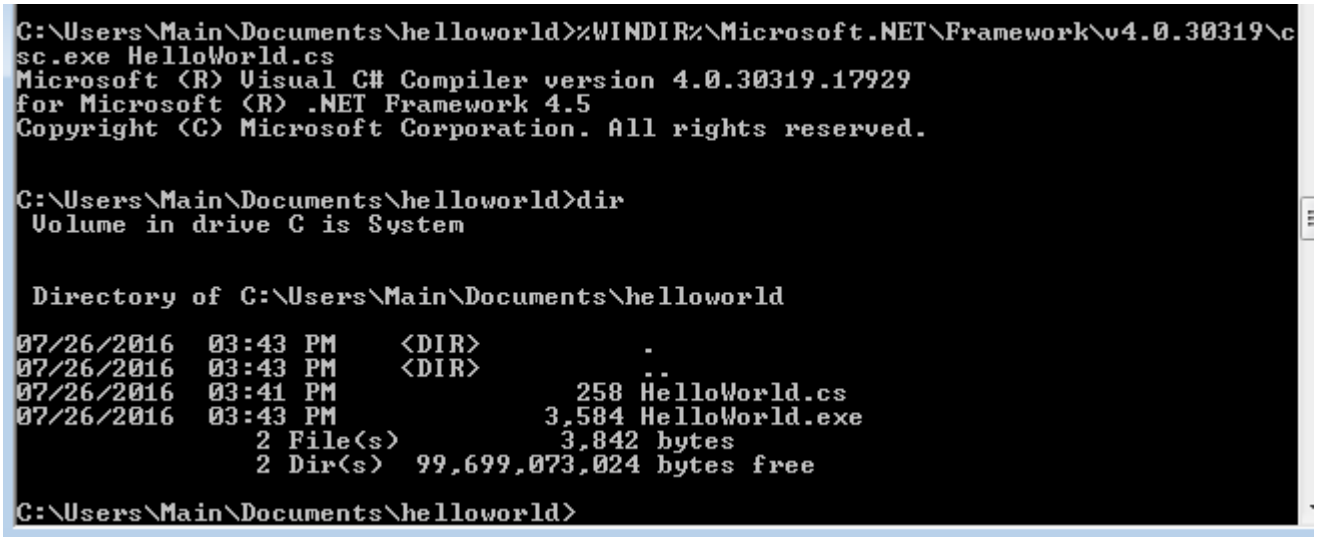
seguente comando nella **console.** (Supponiamo classe `ClassA` ha anche un metodo principale nello stesso `HelloWorld.cs` file in HelloWorld namespace)

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs /main:HelloWorld.ClassA
```

dove HelloWorld è namespace

***Nota** : questo è il percorso in cui **.NET Framework v4.0** si trova in generale. Cambia il percorso in base alla tua versione .NET. Inoltre, la directory potrebbe essere **framework** anziché **framework64** se si utilizza .NET Framework a 32 bit. Dal prompt dei comandi di Windows, è possibile elencare tutti i percorsi del framework csc.exe eseguendo i seguenti comandi (il primo per i framework a 32 bit):*

```
dir %WINDIR%\Microsoft.NET\Framework\csc.exe /s/b
dir %WINDIR%\Microsoft.NET\Framework64\csc.exe /s/b
```



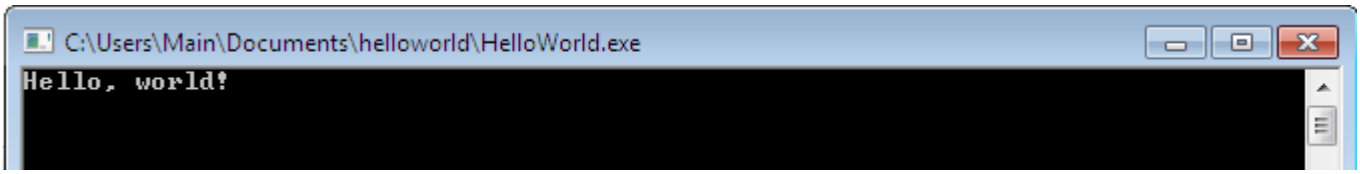
Ora dovrebbe esserci un file eseguibile denominato `HelloWorld.exe` nella stessa directory. Per eseguire il programma dal prompt dei comandi, digita semplicemente il nome del file eseguibile e premi `Invio` come segue:

```
HelloWorld.exe
```

Questo produrrà:

> Ciao mondo!

È anche possibile fare doppio clic sul file eseguibile e avviare una nuova finestra della console con il messaggio " **Hello, world!** "

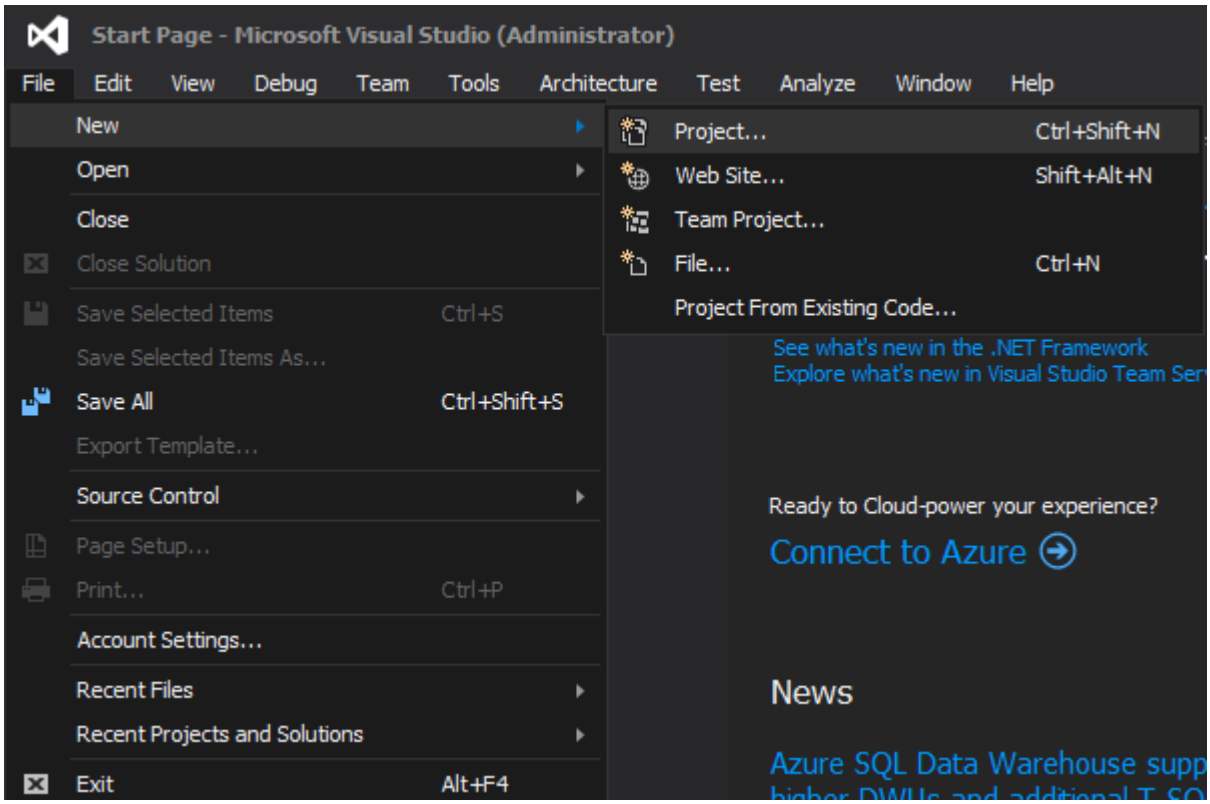


Creazione di un nuovo progetto in Visual Studio (applicazione console) ed esecuzione in modalità Debug

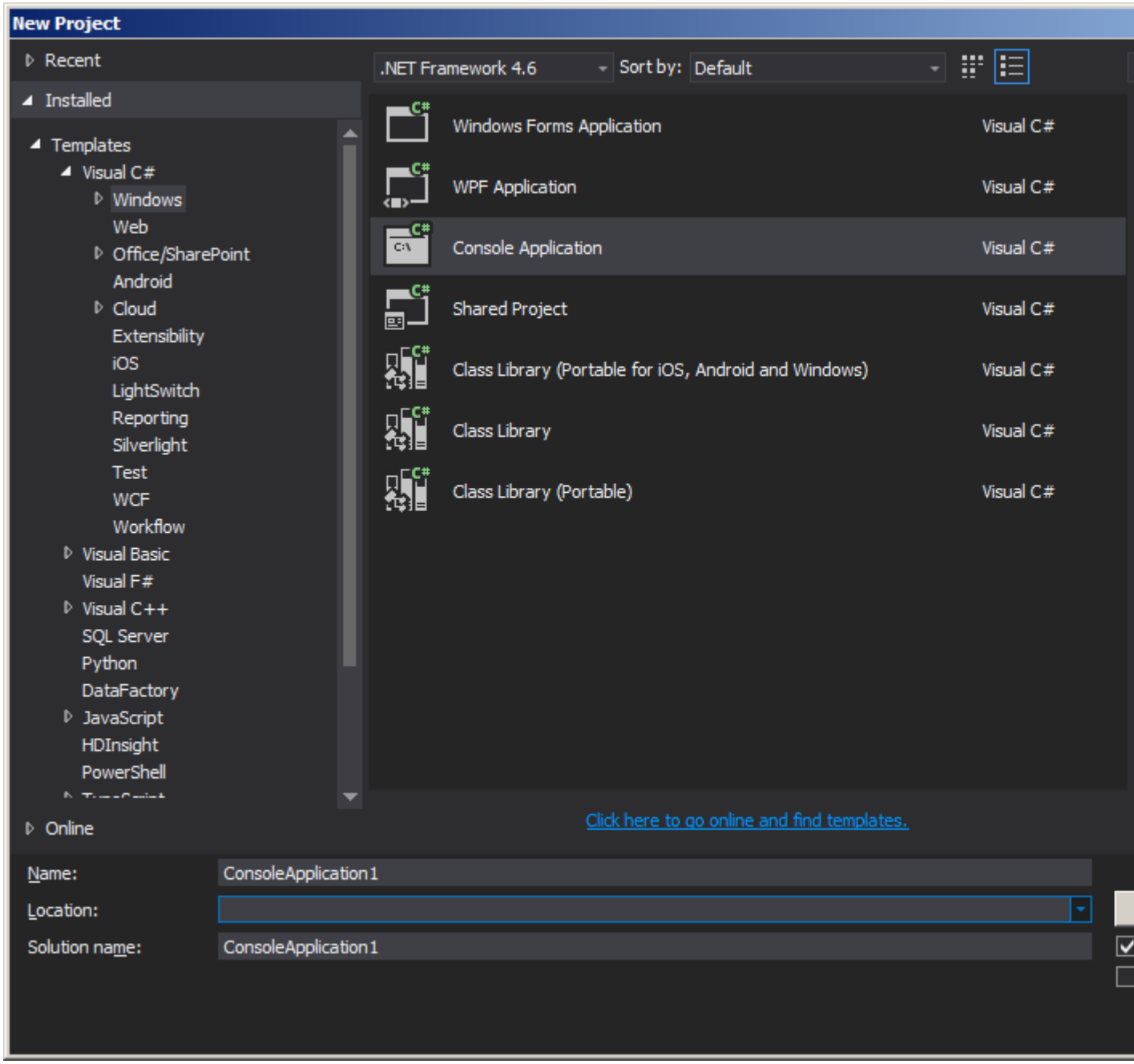
1. **Scarica e installa Visual Studio** . Visual Studio può essere scaricato da VisualStudio.com . L'edizione comunitaria è suggerita, in primo luogo perché è gratuita e in secondo luogo perché include tutte le funzionalità generali e può essere estesa ulteriormente.

2. **Apri Visual Studio.**

3. **Benvenuto.** Vai a **File → Nuovo → Progetto** .

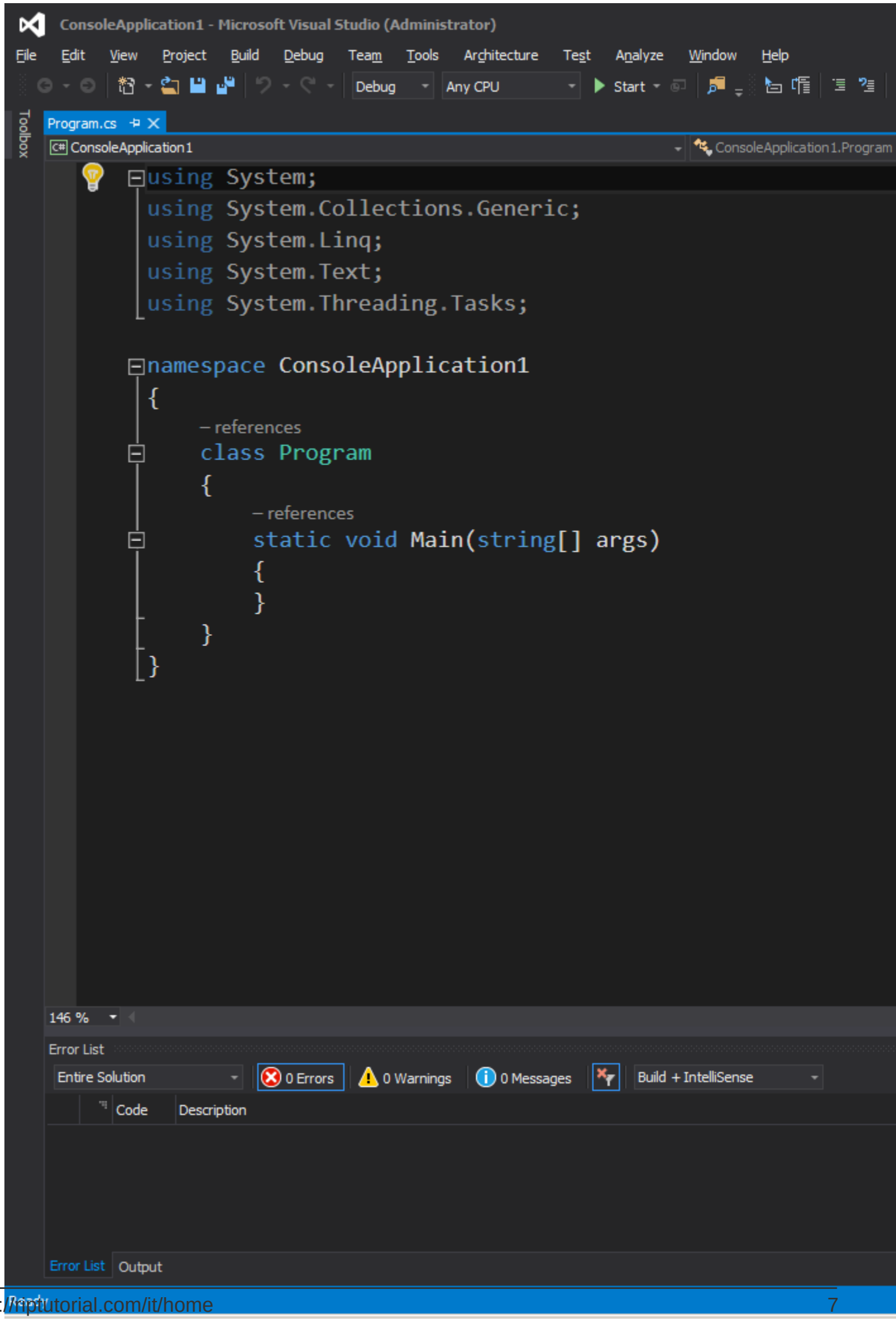


4. Fare clic su **Modelli** → **Visual C #** → **Applicazione console**



5. **Dopo aver selezionato Applicazione console,** immettere un nome per il progetto e una posizione da salvare e premere `OK` . Non preoccuparti del nome della soluzione.

6. **Progetto creato** . Il progetto appena creato sarà simile a:


ConsoleApplication1 - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Debug
Any CPU
Start
Toolbox
Program.cs
ConsoleApplication1
ConsoleApplication1.Program
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ConsoleApplication1
{
0 references
class Program
{
0 references
static void Main(string[] args)
{
}
}
}
146 %
Error List
Entire Solution
0 Errors
0 Warnings
0 Messages
Build + IntelliSense
Code
Description
Error List
Output
Ready

(Utilizzare sempre nomi descrittivi per i progetti in modo che possano essere facilmente distinti da altri progetti. Si consiglia di non utilizzare spazi nel nome del progetto o della classe.)

7. **Scrivi il codice** Ora puoi aggiornare `Program.cs` per presentare "Hello world!" per l'utente.

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

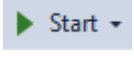
Aggiungi le seguenti due righe all'oggetto `public static void Main(string[] args)` in `Program.cs` : (assicurati che sia all'interno delle parentesi graffe)

`Program.cs` : (assicurati che sia all'interno delle parentesi graffe)

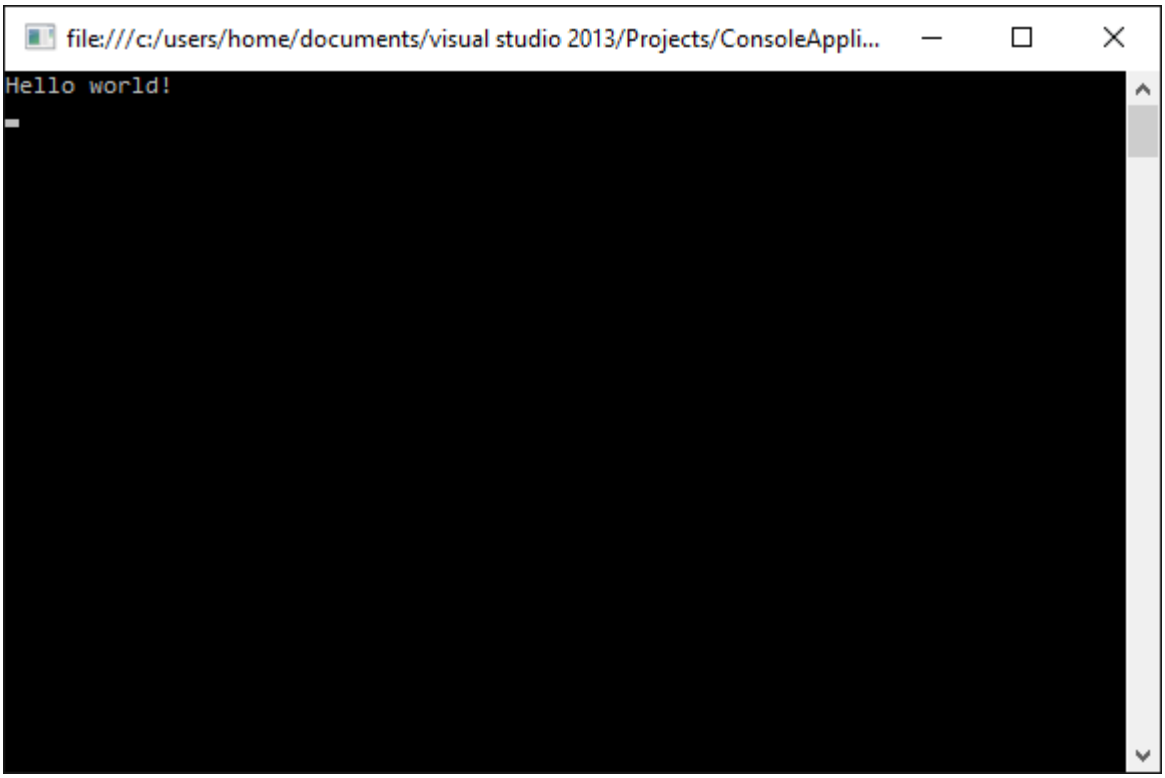
```
Console.WriteLine("Hello world!");
Console.Read();
```

Perché `Console.Read()` **?** La prima riga stampa il testo "Hello world!" alla console, e la seconda linea attende l'immissione di un singolo carattere; in effetti, questo fa in modo che il programma interrompa l'esecuzione in modo che tu possa vedere l'output durante il debug. Senza `Console.Read();` , quando avvii il debug dell'applicazione, verrà semplicemente stampato "Hello world!" alla console e quindi chiudere immediatamente. La finestra del codice dovrebbe ora apparire come la seguente:

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
            Console.Read();
        }
    }
}
```

8. **Esegui il debug del tuo programma.** Premi il pulsante Start sulla barra degli strumenti vicino alla parte superiore della finestra 

o premere `F5` sulla tastiera per eseguire l'applicazione. Se il pulsante non è presente, è possibile eseguire il programma dal menu principale: **Debug → Avvia debug** . Il programma compilerà e quindi aprirà una finestra della console. Dovrebbe apparire simile allo screenshot seguente:



9. **Ferma il programma.** Per chiudere il programma, basta premere un tasto qualsiasi sulla tastiera. La `Console.Read()` abbiamo aggiunto era per lo stesso scopo. Un altro modo per chiudere il programma è andare nel menu in cui si trovava il pulsante `Start` e fare clic sul pulsante `Stop` .

Creare un nuovo programma usando Mono

Innanzitutto installa Mono seguendo le istruzioni di installazione per la piattaforma scelta come descritto nella relativa sezione di installazione .

Mono è disponibile per Mac OS X, Windows e Linux.

Al termine dell'installazione, creare un file di testo, denominarlo `HelloWorld.cs` e copiare il seguente contenuto in esso:

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, world!");
        System.Console.WriteLine("Press any key to exit..");
        System.Console.Read();
    }
}
```

Se si utilizza Windows, eseguire il Prompt dei comandi mono incluso nell'installazione di Mono e assicurarsi che siano impostate le variabili di ambiente necessarie. Se su Mac o Linux, apri un nuovo terminale.

Per compilare il file appena creato, eseguire il seguente comando nella directory contenente `HelloWorld.cs`

:

```
mcs -out:HelloWorld.exe HelloWorld.cs
```

Il risultato `HelloWorld.exe` può quindi essere eseguito con:

```
mono HelloWorld.exe
```

che produrrà l'output:

```
Hello, world!
Press any key to exit..
```

Creazione di un nuovo programma utilizzando .NET Core

Innanzitutto installa **.NET Core SDK** seguendo le istruzioni di installazione per la piattaforma di tua scelta:

- finestre
- OSX
- Linux
- docker

Al termine dell'installazione, aprire un prompt dei comandi o una finestra di terminale.

1. Crea una nuova directory con `mkdir hello_world` e cambia nella directory appena creata con `cd hello_world` .

2. Crea una nuova applicazione console con `dotnet new console` .
   Questo produrrà due file:

   - **hello_world.csproj**

     ```
     <Project Sdk="Microsoft.NET.Sdk">

       <PropertyGroup>
         <OutputType>Exe</OutputType>
         <TargetFramework>netcoreapp1.1</TargetFramework>
       </PropertyGroup>

     </Project>
     ```

   - **Program.cs**

     ```
     using System;

     namespace hello_world
     {
         class Program
         {
             static void Main(string[] args)
     ```

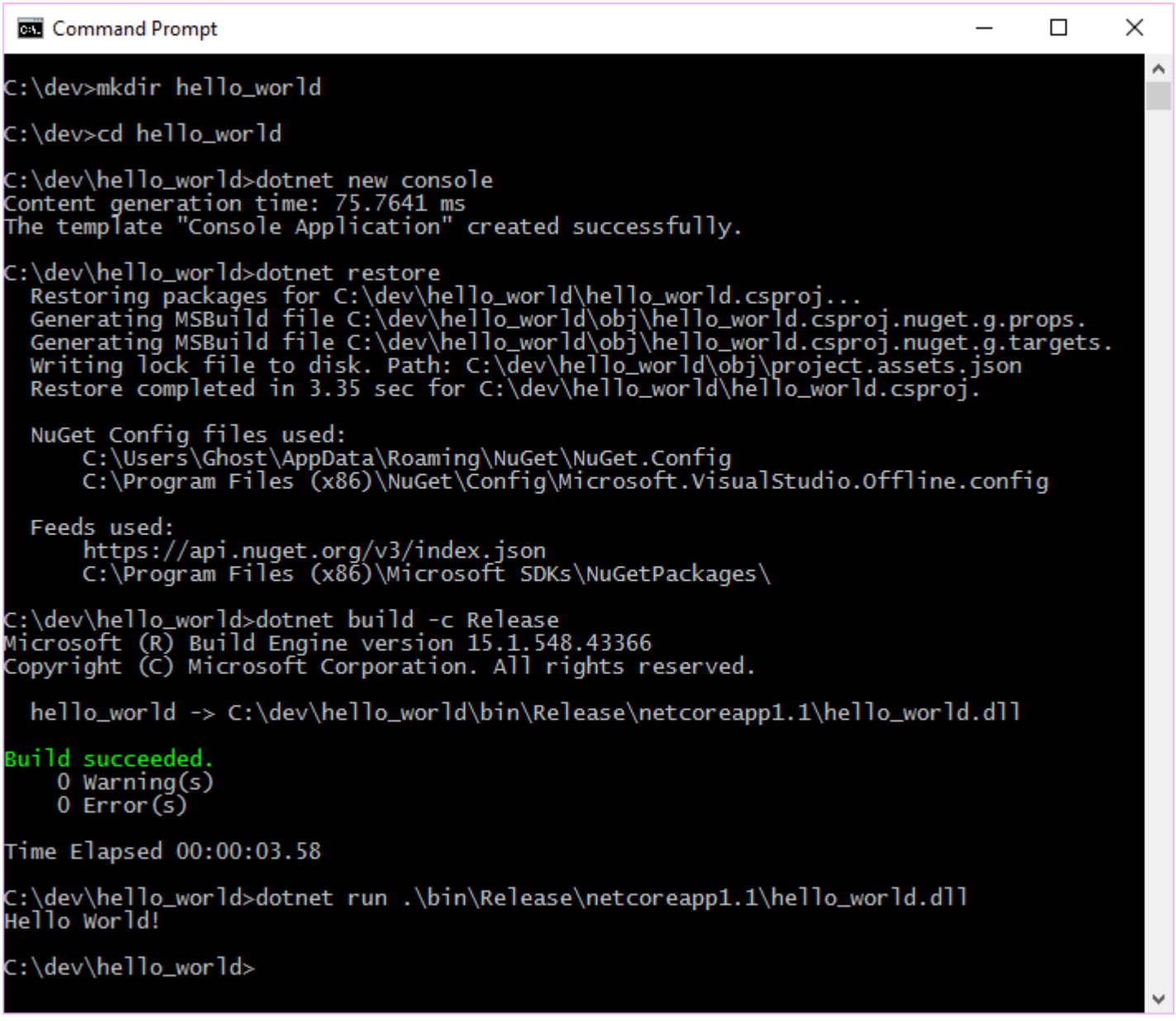
```
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

3. Ripristina i pacchetti necessari con `dotnet restore` .

4. *Facoltativo* Crea l'applicazione con `dotnet build` per Debug o `dotnet build -c Release` for Release. `dotnet run` eseguirà anche il compilatore e genererà errori di compilazione, se presenti.

5. Esegui l'applicazione con `dotnet run` per `dotnet run` debug o `dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll` per la versione.

Output del prompt dei comandi

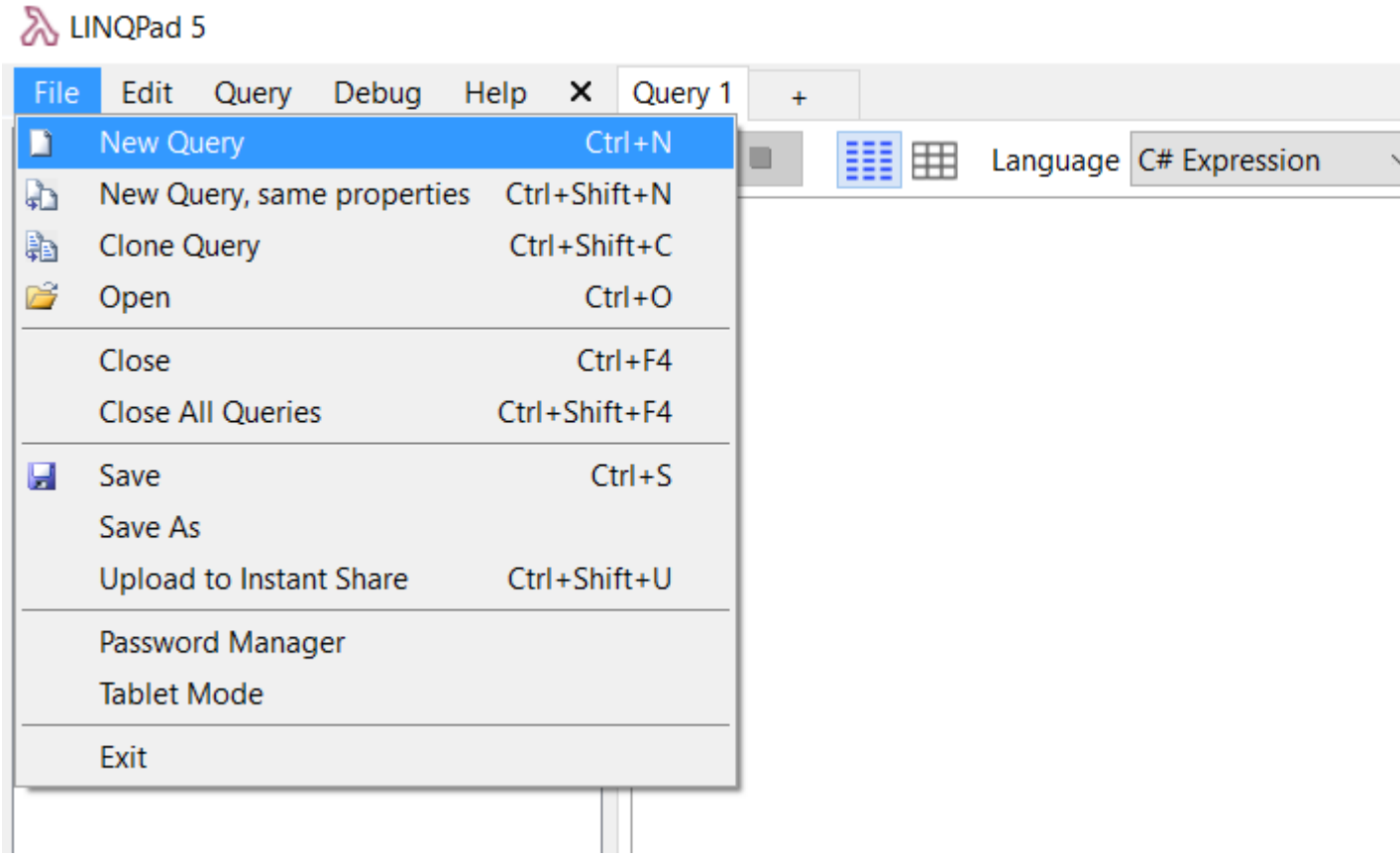


Creare una nuova query usando LinqPad

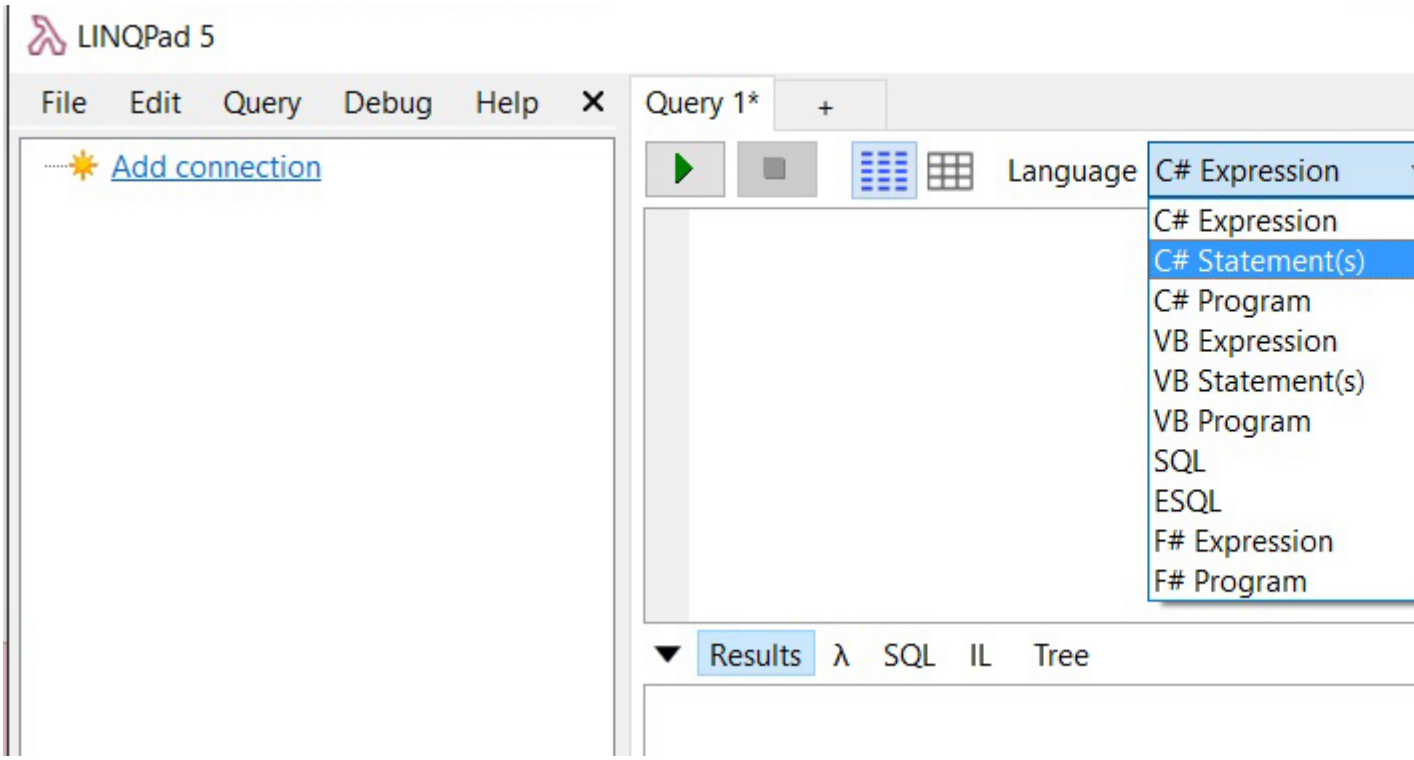
LinqPad è un ottimo strumento che ti consente di apprendere e testare le funzionalità dei linguaggi .Net (C #, F # e VB.Net.)

1. Installa LinqPad

2. Crea una nuova query (`Ctrl` + `N`)

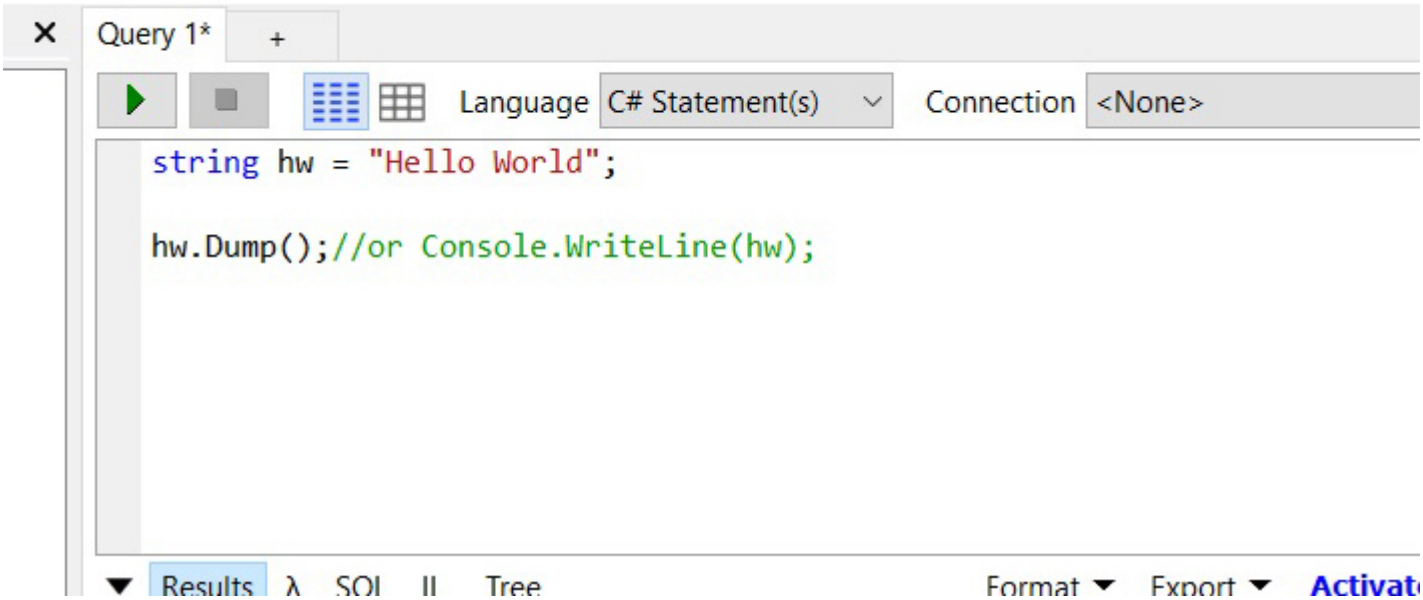


3. Sotto la lingua, seleziona "Dichiarazioni C #"

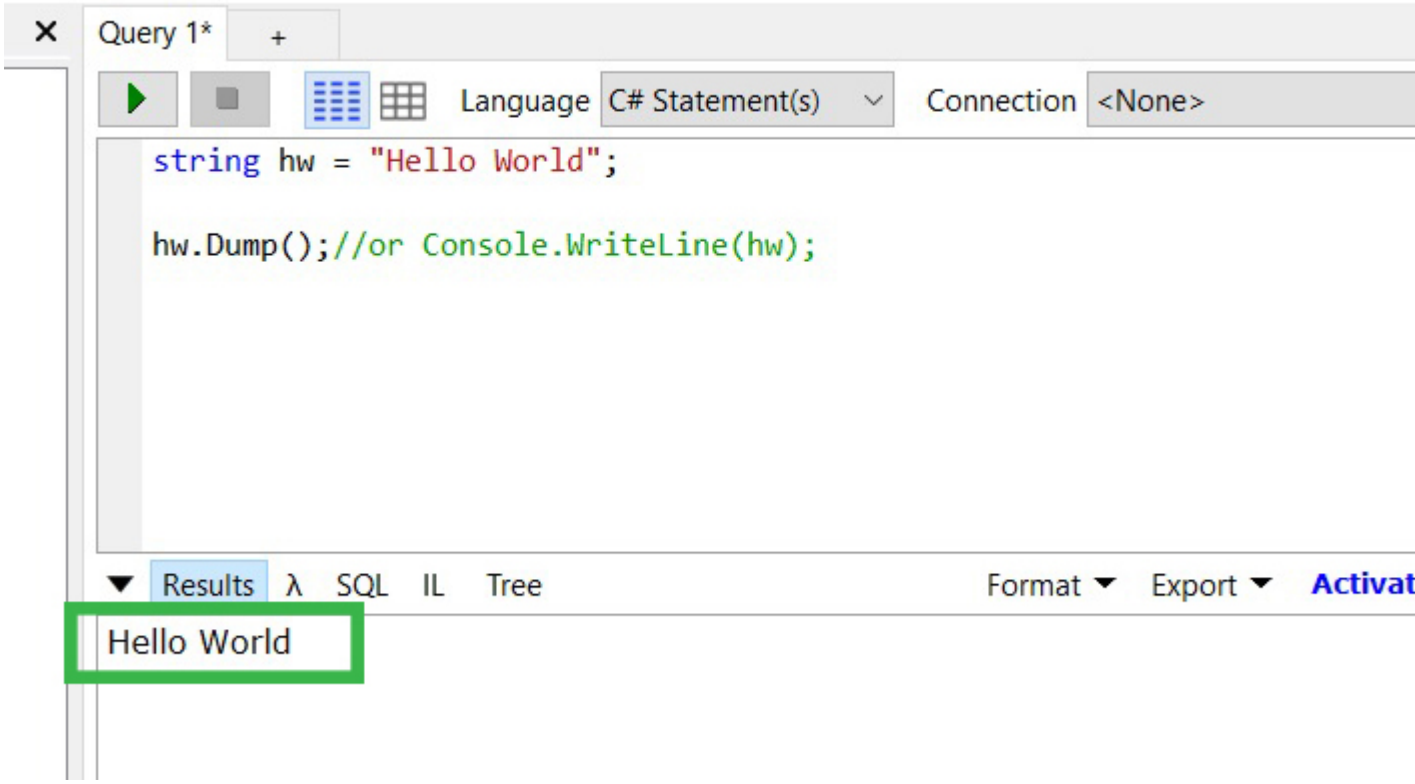


4. Digita il codice seguente e premi run (`F5`)

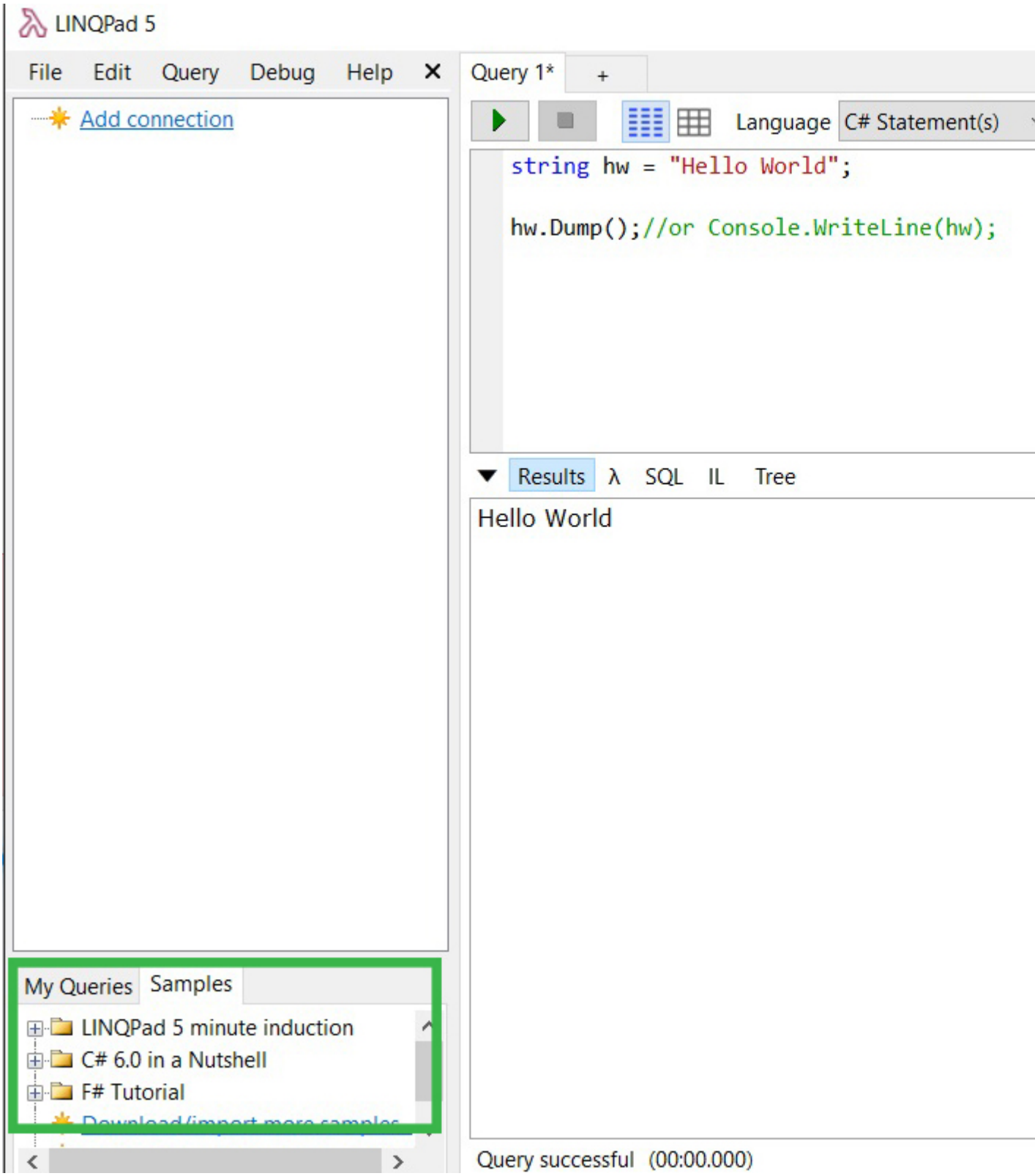
```
string hw = "Hello World";

hw.Dump(); //or Console.WriteLine(hw);
```



5. Dovresti vedere "Hello World" stampato nella schermata dei risultati.

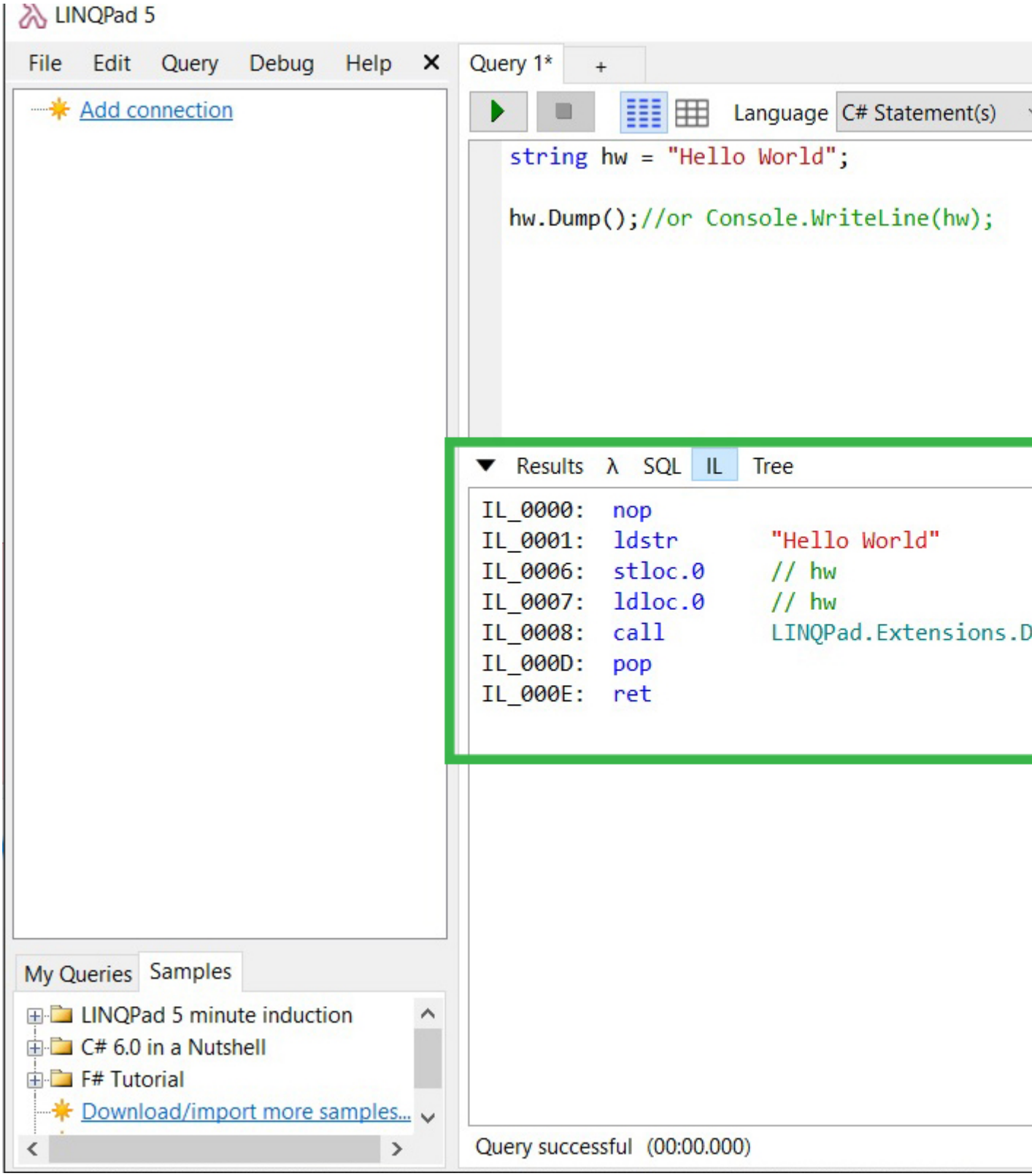


6. Ora che hai creato il tuo primo programma .Net, vai e controlla gli esempi inclusi in LinqPad tramite il browser "Samples". Ci sono molti ottimi esempi che ti mostreranno molte caratteristiche differenti dei linguaggi .Net.



Gli appunti:

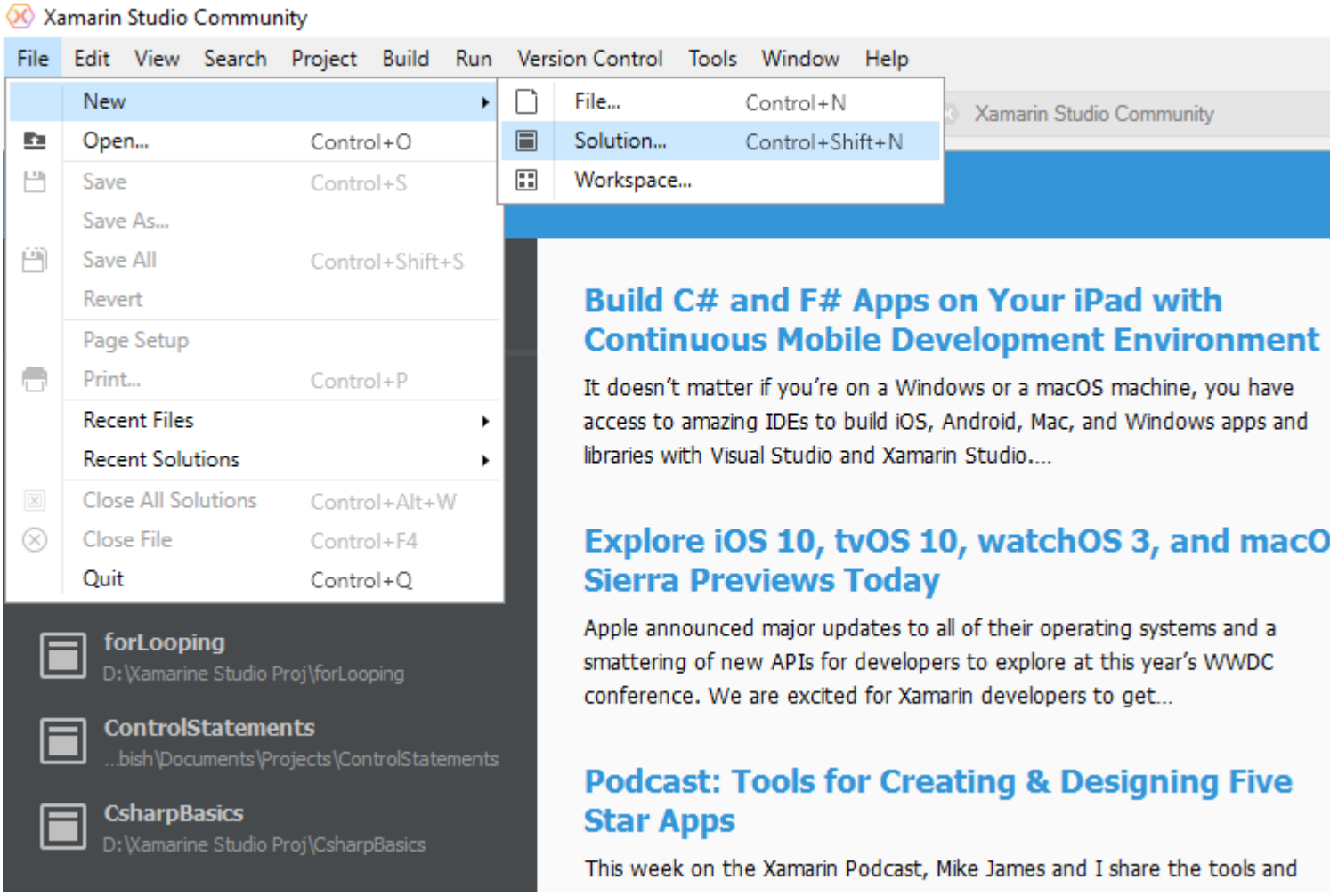
1. Se si fa clic su "IL", è possibile controllare il codice IL generato dal codice .net. Questo è un grande strumento di apprendimento.



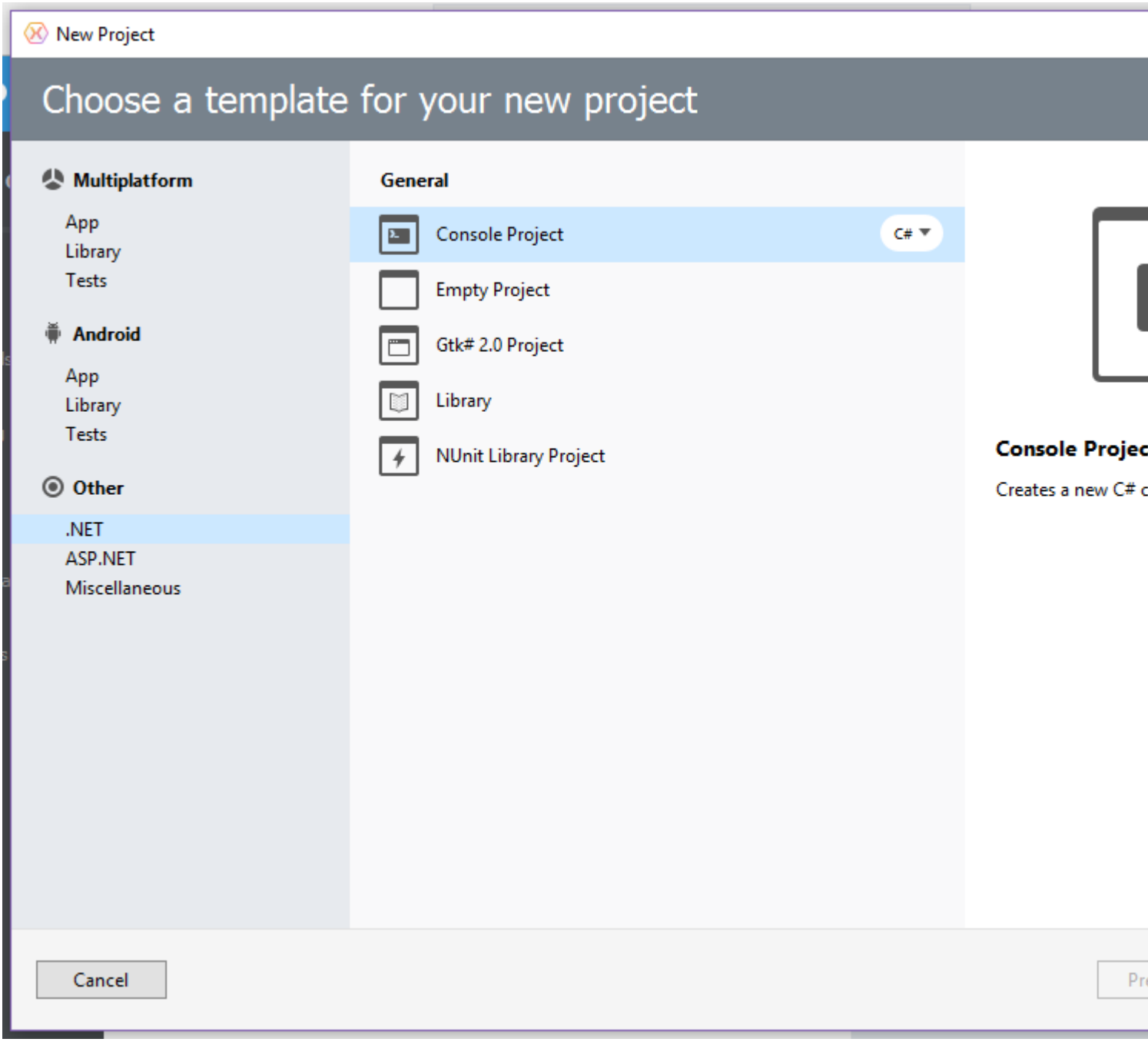
2. Quando si utilizza `LINQ to SQL` o `Linq to Entities` è possibile esaminare l'SQL generato, un altro ottimo modo per conoscere LINQ.

Creare un nuovo progetto usando Xamarin Studio

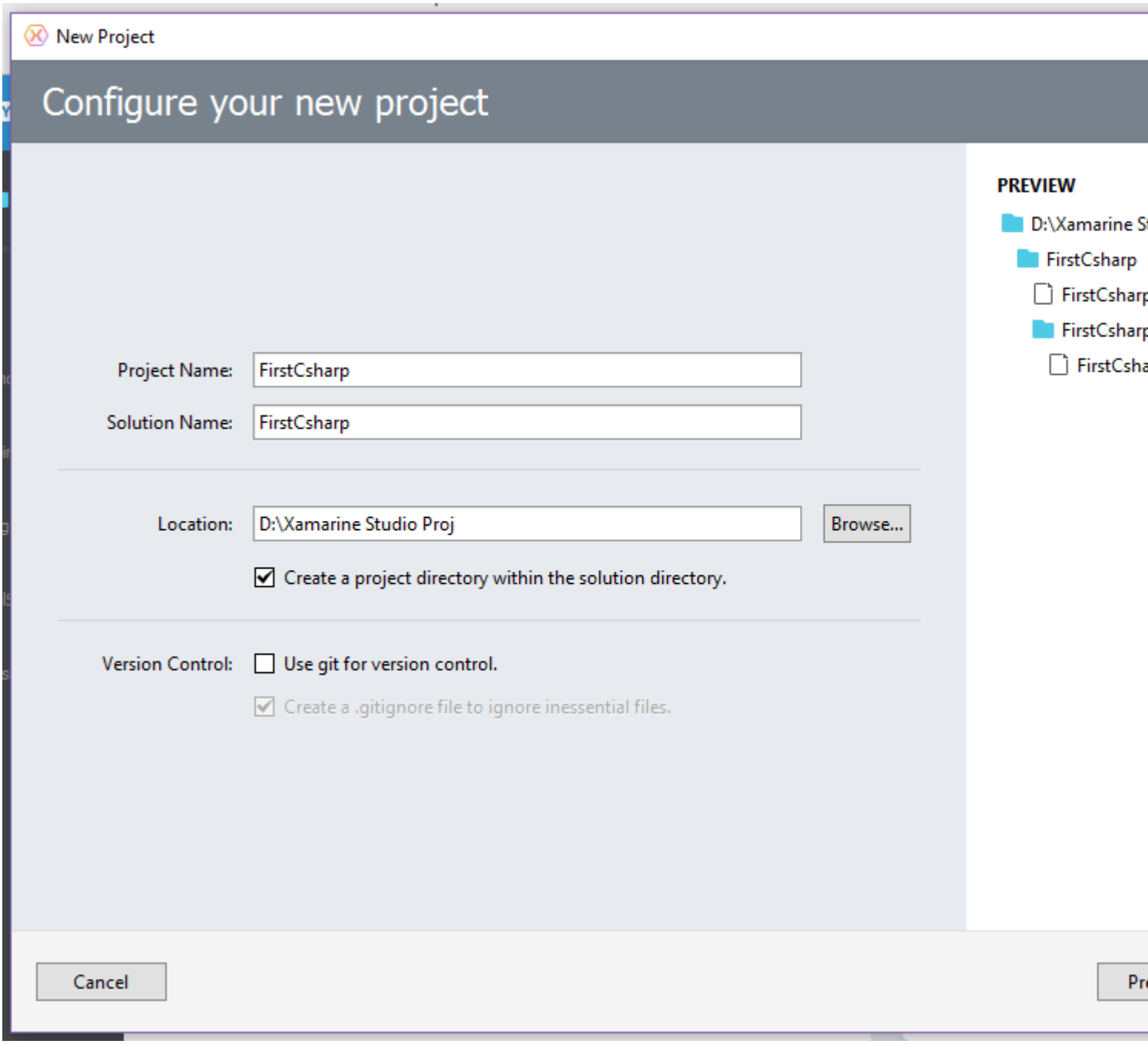
1. Scarica e installa la community di Xamarin Studio .
2. Apri Xamarin Studio.
3. Fare clic su **File** → **Nuovo** → **Soluzione** .



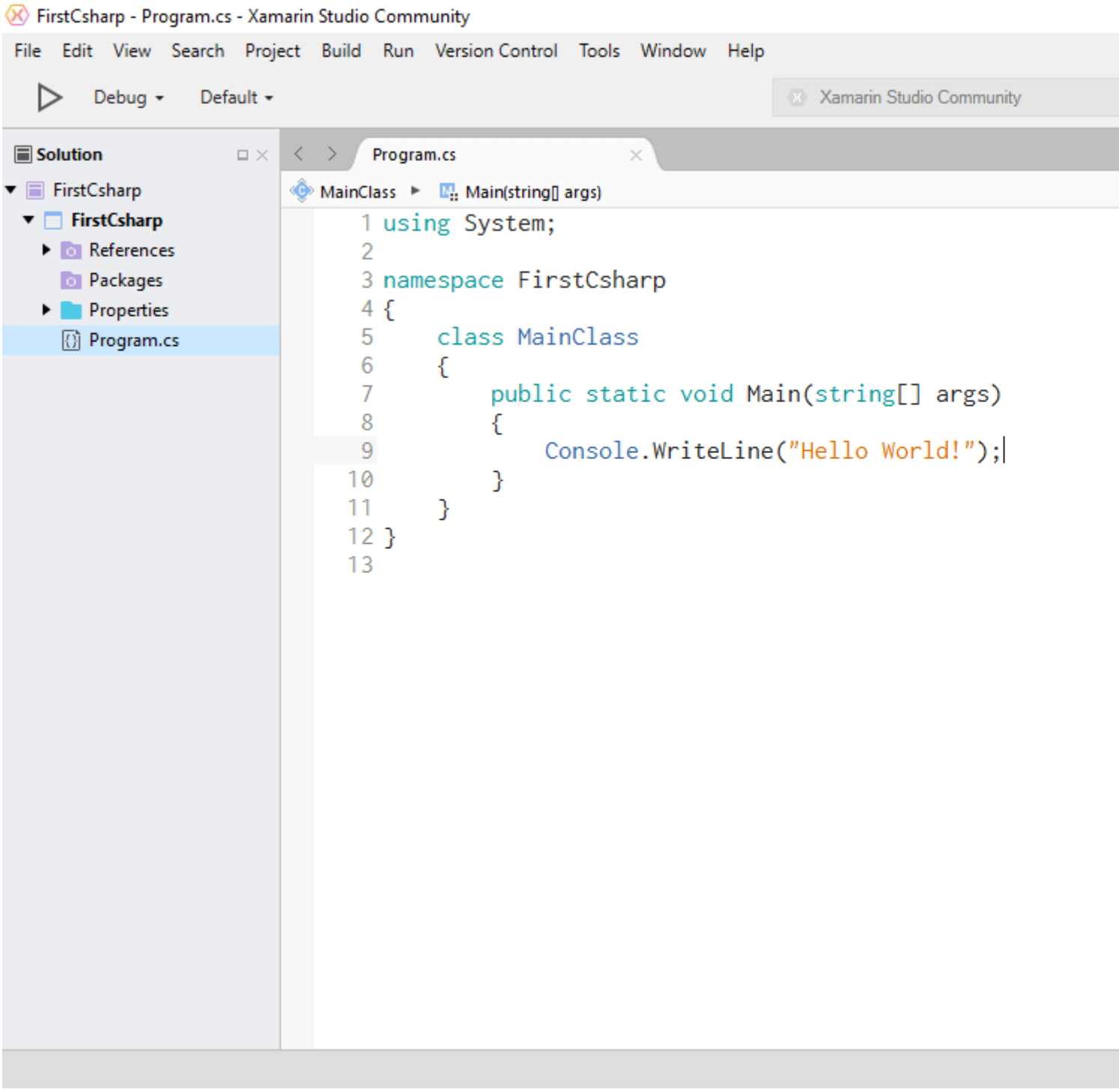
4. Fare clic su **.NET** → **Progetto console** e selezionare **C #** .
5. Fare clic su `Avanti` per procedere.



6. Immettere il **nome** del **progetto** e `Sfoglia ...` per una **posizione** da salvare, quindi fare clic su `Crea` .



7. Il progetto appena creato sarà simile a:

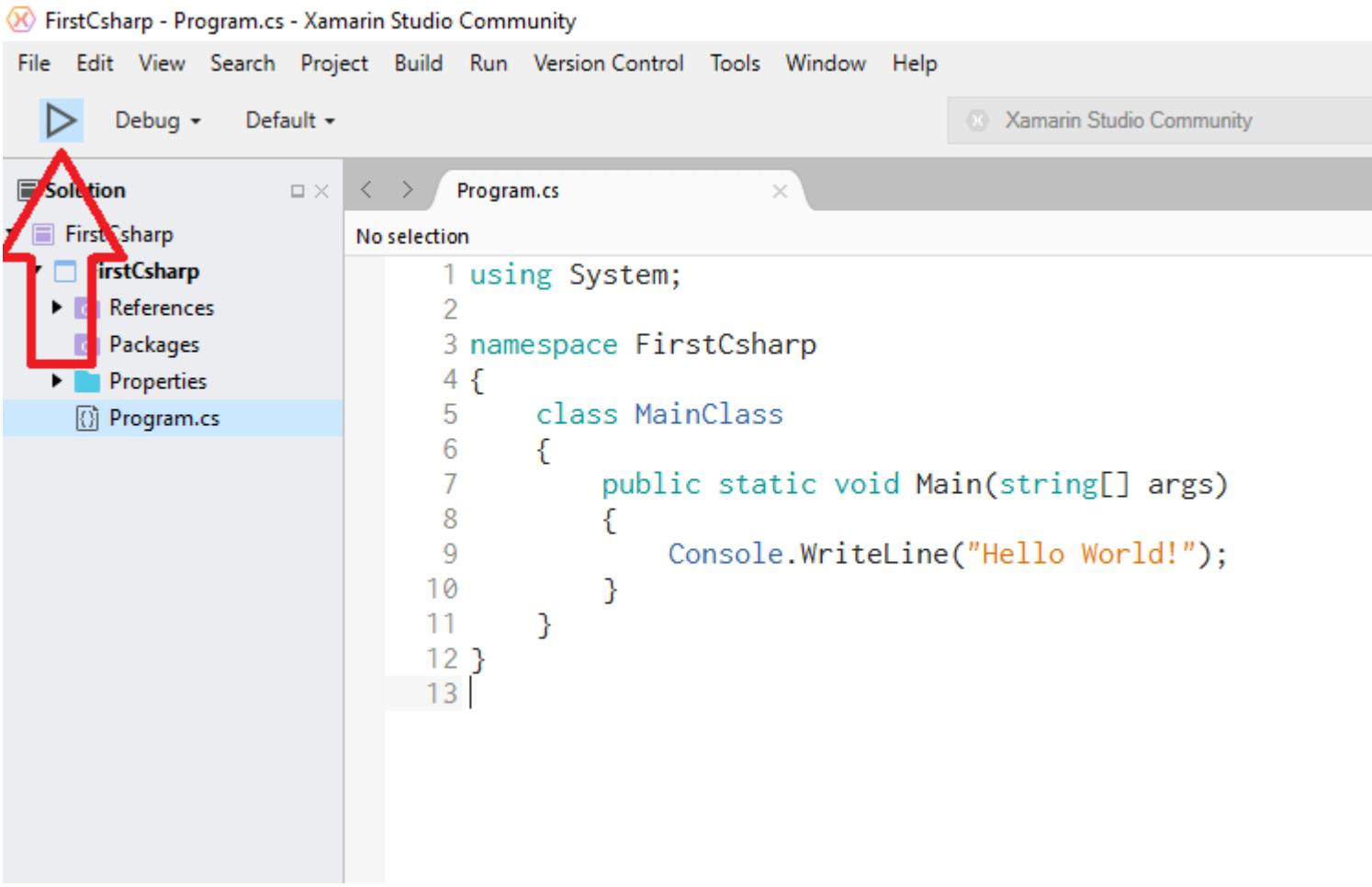


8. Questo è il codice nell'editor di testo:

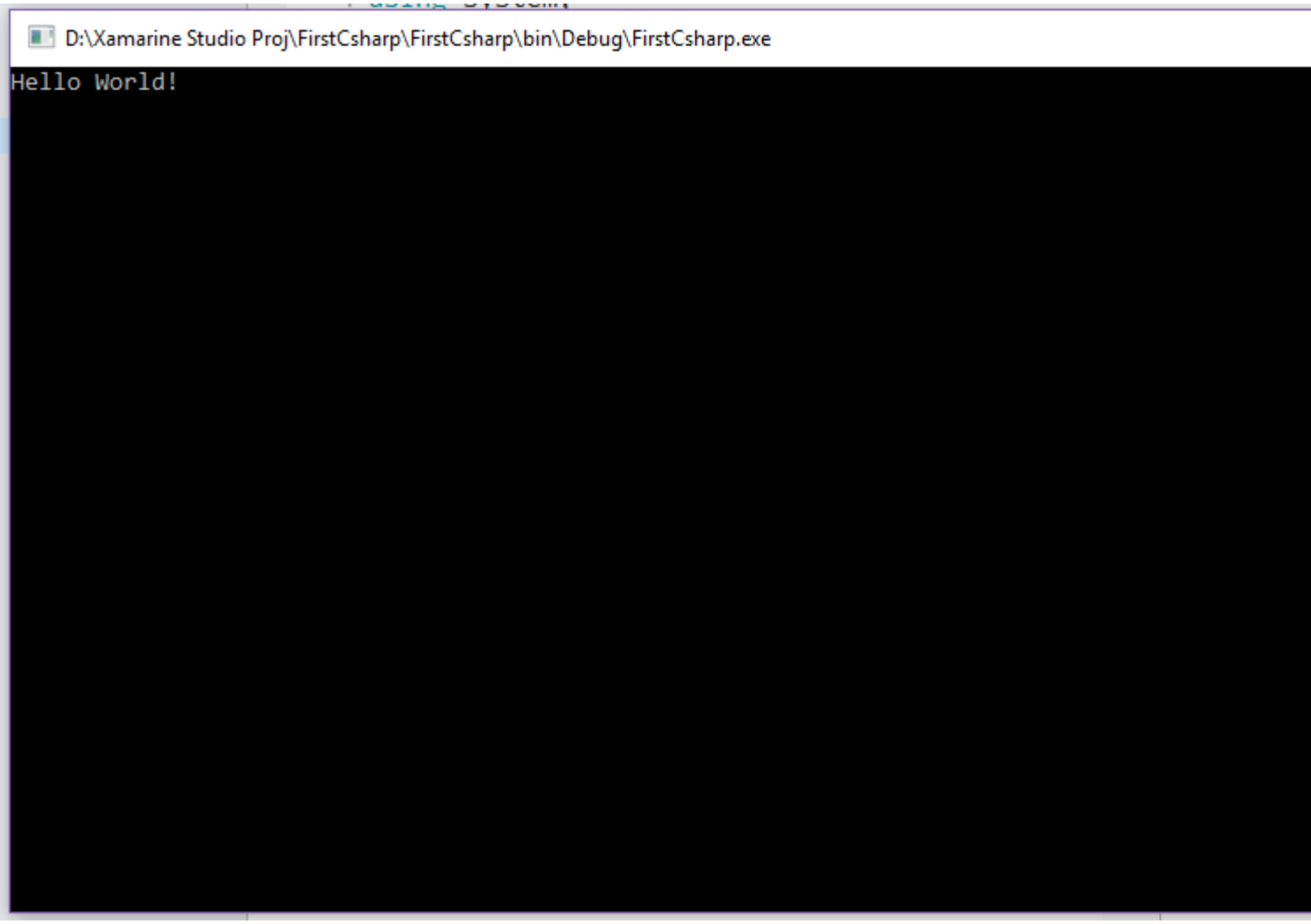
```
using System;

namespace FirstCsharp
{
    public class MainClass
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
```

```
}
```

9. Per eseguire il codice, premi `F5` o fai clic sul **pulsante Riproduci** come mostrato di seguito:



10. Di seguito è riportato l'output:



Leggi Iniziare con C # Language online: https://riptutorial.com/it/csharp/topic/15/iniziare-con-c-sharp-language

Capitolo 2: .NET Compiler Platform (Roslyn)

Examples

Crea spazio di lavoro dal progetto MSBuild

Prima di continuare, ottenere prima il nuget `Microsoft.CodeAnalysis.CSharp.Workspaces` .

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var project = await workspace.OpenProjectAsync(projectFilePath);
var compilation = await project.GetCompilationAsync();

foreach (var diagnostic in compilation.GetDiagnostics()
    .Where(d => d.Severity == Microsoft.CodeAnalysis.DiagnosticSeverity.Error))
{
    Console.WriteLine(diagnostic);
}
```

Per caricare il codice esistente nello spazio di lavoro, compilare e segnalare errori. Successivamente il codice sarà localizzato in memoria. Da qui, saranno disponibili sia il lato sintattico che quello semantico con cui lavorare.

Albero della sintassi

Un **Syntax Tree** è una struttura di dati immutabile che rappresenta il programma come un albero di nomi, comandi e segni (come precedentemente configurato nell'editor).

Ad esempio, si supponga che sia stata configurata un'istanza `Microsoft.CodeAnalysis.Compilation` denominata `compilation` . Esistono diversi modi per elencare i nomi di ogni variabile dichiarata nel codice caricato. Per farlo in modo ingenuo, prendi tutti i pezzi di sintassi in ogni documento (il metodo `DescendantNodes`) e usa Linq per selezionare i nodi che descrivono la dichiarazione delle variabili:

```
foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();
    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>()
        .Select(vd => vd.Identifier);

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di);
    }
}
```

Nell'edificio della sintassi esisteranno tutti i tipi di costrutti C # con un tipo corrispondente. Per trovare rapidamente tipi specifici, utilizzare la finestra `Syntax Visualizer` di Visual Studio. Questo interpreterà il documento aperto corrente come un albero di sintassi di Roslyn.

Modello semantico

Un **modello semantico** offre un livello più profondo di interpretazione e intuizione del codice rispetto a un albero di sintassi. Laddove gli alberi di sintassi possono indicare i nomi delle variabili, i modelli semantici forniscono anche il tipo e tutti i riferimenti. Le chiamate del metodo di avviso degli alberi di sintassi, ma i modelli semantici forniscono riferimenti alla posizione precisa in cui è stato dichiarato il metodo (dopo che è stata applicata la risoluzione di sovraccarico).

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var sln = await workspace.OpenSolutionAsync(solutionFilePath);
var project = sln.Projects.First();
var compilation = await project.GetCompilationAsync();

foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();

    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>();

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di.Identifier);
        // => "root"

        var variableSymbol = compilation
            .GetSemanticModel(syntaxTree)
            .GetDeclaredSymbol(di) as ILocalSymbol;

        Console.WriteLine(variableSymbol.Type);
        // => "Microsoft.CodeAnalysis.SyntaxNode"

        var references = await SymbolFinder.FindReferencesAsync(variableSymbol, sln);
        foreach (var reference in references)
        {
            foreach (var loc in reference.Locations)
            {
                Console.WriteLine(loc.Location.SourceSpan);
                // => "[1375..1379)"
            }
        }
    }
}
```

Questo produce un elenco di variabili locali usando un albero di sintassi. Quindi consulta il modello semantico per ottenere il nome completo del tipo e trova tutti i riferimenti di ogni variabile.

Leggi .NET Compiler Platform (Roslyn) online: https://riptutorial.com/it/csharp/topic/4886/-net-compiler-platform--roslyn-

Capitolo 3: Accedi alla cartella condivisa di rete con nome utente e password

introduzione

Accesso al file di condivisione di rete tramite PInvoke.

Examples

Codice per accedere al file condiviso di rete

```
public class NetworkConnection : IDisposable
    {
        string _networkName;

        public NetworkConnection(string networkName,
            NetworkCredential credentials)
        {
            _networkName = networkName;

            var netResource = new NetResource()
            {
                Scope = ResourceScope.GlobalNetwork,
                ResourceType = ResourceType.Disk,
                DisplayType = ResourceDisplaytype.Share,
                RemoteName = networkName
            };

            var userName = string.IsNullOrEmpty(credentials.Domain)
                ? credentials.UserName
                : string.Format(@"{0}\{1}", credentials.Domain, credentials.UserName);

            var result = WNetAddConnection2(
                netResource,
                credentials.Password,
                userName,
                0);

            if (result != 0)
            {
                throw new Win32Exception(result);
            }
        }

    ~NetworkConnection()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
```

```
    }

    protected virtual void Dispose(bool disposing)
    {
        WNetCancelConnection2(_networkName, 0, true);
    }

    [DllImport("mpr.dll")]
    private static extern int WNetAddConnection2(NetResource netResource,
        string password, string username, int flags);

    [DllImport("mpr.dll")]
    private static extern int WNetCancelConnection2(string name, int flags,
        bool force);
}

[StructLayout(LayoutKind.Sequential)]
public class NetResource
{
    public ResourceScope Scope;
    public ResourceType ResourceType;
    public ResourceDisplaytype DisplayType;
    public int Usage;
    public string LocalName;
    public string RemoteName;
    public string Comment;
    public string Provider;
}

public enum ResourceScope : int
{
    Connected = 1,
    GlobalNetwork,
    Remembered,
    Recent,
    Context
};

public enum ResourceType : int
{
    Any = 0,
    Disk = 1,
    Print = 2,
    Reserved = 8,
}

public enum ResourceDisplaytype : int
{
    Generic = 0x0,
    Domain = 0x01,
    Server = 0x02,
    Share = 0x03,
    File = 0x04,
    Group = 0x05,
    Network = 0x06,
    Root = 0x07,
    Shareadmin = 0x08,
    Directory = 0x09,
    Tree = 0x0a,
    Ndscontainer = 0x0b
}
```

Leggi Accedi alla cartella condivisa di rete con nome utente e password online: https://riptutorial.com/it/csharp/topic/9627/accedi-alla-cartella-condivisa-di-rete-con-nome-utente-e-password

Capitolo 4: Accesso ai database

Examples

Connessioni ADO.NET

Le connessioni ADO.NET sono uno dei modi più semplici per connettersi a un database da un'applicazione C #. Si basano sull'utilizzo di un provider e di una stringa di connessione che punta al database per eseguire query contro.

Classi di provider di dati comuni

Molte delle seguenti sono classi che vengono comunemente utilizzate per interrogare i database e i relativi spazi dei nomi:

- `SqlConnection` , `SqlCommand` , `SqlDataReader` di `System.Data.SqlClient`
- `OleDbConnection` , `OleDbCommand` , `OleDbDataReader` di `System.Data.OleDb`
- `MySqlConnection` , `MySqlCommand` , `MySqlDbDataReader` di `MySql.Data`

Tutti questi sono comunemente usati per accedere ai dati tramite C # e verranno comunemente riscontrati durante la creazione di applicazioni incentrate sui dati. `FooDataReader` si può aspettare che molte altre classi che non sono menzionate che implementano le stesse `FooConnection` , `FooCommand` , `FooDataReader` si comportino allo stesso modo.

Modello di accesso comune per connessioni ADO.NET

Un modello comune che può essere utilizzato quando si accede ai dati tramite una connessione ADO.NET potrebbe essere il seguente:

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}")
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property");
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
         // Open your connection
         connection.Open();

         // Add your parameters here if necessary

         // Execute your query as a reader (again scoped with a using statement)
         using(var reader = command.ExecuteReader())
         {
               // Iterate through your results here
         }
    }
}
```

O se stavi semplicemente eseguendo un semplice aggiornamento e non avessi bisogno di un lettore, si applicherebbe lo stesso concetto di base:

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
     var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
     using(var command = new SqlCommand(query,connection))
     {
          connection.Open();

          // Add parameters here

          // Perform your update
          command.ExecuteNonQuery();
     }
}
```

È anche possibile programmare contro una serie di interfacce comuni e non doversi preoccupare delle classi specifiche del provider. Le interfacce principali fornite da ADO.NET sono:

- IDbConnection - per la gestione delle connessioni del database
- IDbCommand - per l'esecuzione di comandi SQL
- IDbTransaction - per la gestione delle transazioni
- IDataReader - per leggere i dati restituiti da un comando
- IDataAdapter - per il channeling dei dati da e verso i set di dati

```
var connectionString = "{your-connection-string}";
var providerName = "{System.Data.SqlClient}"; //for Oracle use
"Oracle.ManagedDataAccess.Client"
//most likely you will get the above two from ConnectionStringSettings object

var factory = DbProviderFactories.GetFactory(providerName);

using(var connection = new factory.CreateConnection()) {
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = new connection.CreateCommand()) {
        command.CommandText = "{sql-query}";    //this needs to be tailored for each database
system

        using(var reader = command.ExecuteReader()) {
            while(reader.Read()) {
                ...
            }
        }
    }
}
```

Entity Framework Connections

Entity Framework espone le classi di astrazione utilizzate per interagire con i database sottostanti sotto forma di classi come `DbContext` . Questi contesti generalmente sono costituiti da proprietà `DbSet<T>` che espongono le raccolte disponibili che possono essere interrogate:

```
public class ExampleContext: DbContext
{
    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

Lo stesso `DbContext` gestirà le connessioni con i database e in genere leggerà i dati della stringa di connessione appropriata da una configurazione per determinare come stabilire le connessioni:

```
public class ExampleContext: DbContext
{
    // The parameter being passed in to the base constructor indicates the name of the
    // connection string
    public ExampleContext() : base("ExampleContextEntities")
    {
    }

    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

Esecuzione di query di Entity Framework

L'esecuzione effettiva di una query di Entity Framework può essere abbastanza semplice e richiede semplicemente di creare un'istanza del contesto e quindi utilizzare le proprietà disponibili su di essa per estrarre o accedere ai dati

```
using(var context = new ExampleContext())
{
      // Retrieve all of the Widgets in your database
      var data = context.Widgets.ToList();
}
```

Entity Framework fornisce inoltre un ampio sistema di tracciabilità delle modifiche che può essere utilizzato per gestire le voci di aggiornamento all'interno del database semplicemente chiamando il metodo `SaveChanges()` per inviare modifiche al database:

```
using(var context = new ExampleContext())
{
      // Grab the widget you wish to update
      var widget = context.Widgets.Find(w => w.Id == id);
      // If it exists, update it
      if(widget != null)
      {
           // Update your widget and save your changes
           widget.Updated = DateTime.UtcNow;
           context.SaveChanges();
      }
}
```

Stringhe di connessione

Una stringa di connessione è una stringa che specifica le informazioni su una particolare fonte di dati e come procedere per connettersi ad essa memorizzando credenziali, posizioni e altre

informazioni.

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

Memorizzazione della stringa di connessione

In genere, una stringa di connessione verrà archiviata in un file di configurazione (come `app.config` o `web.config` all'interno di applicazioni ASP.NET). Di seguito è riportato un esempio di come potrebbe apparire una connessione locale all'interno di uno di questi file:

```
<connectionStrings>
   <add name="WidgetsContext" providerName="System.Data.SqlClient"
connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=True;"/>
</connectionStrings>

<connectionStrings>
   <add name="WidgetsContext" providerName="System.Data.SqlClient"
connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=SSPI;"/>
</connectionStrings>
```

Ciò consentirà alla tua applicazione di accedere alla stringa di connessione in modo programmatico tramite `WidgetsContext` . Sebbene sia `Integrated Security=SSPI` e `Integrated Security=True` svolgono la stessa funzione; `Integrated Security=SSPI` è preferito poiché funziona con entrambi i provider SQLClient e OleDB, dove come `Integrated Security=true` genera un'eccezione quando utilizzato con il provider OleDb.

Connessioni diverse per provider diversi

Ogni fornitore di dati (SQL Server, MySQL, Azure, ecc.) Presenta il proprio aroma di sintassi per le stringhe di connessione ed espone diverse proprietà disponibili. ConnectionStrings.com è una risorsa incredibilmente utile se non sei sicuro di come dovrebbe essere il tuo.

Leggi Accesso ai database online: https://riptutorial.com/it/csharp/topic/4811/accesso-ai-database

Capitolo 5: Alberi di espressione

introduzione

Gli alberi di espressione sono espressioni disposte in una struttura di dati simile a un albero. Ogni nodo nell'albero è una rappresentazione di un'espressione, un'espressione essendo un codice. Una rappresentazione in memoria di un'espressione Lambda sarebbe un albero di espressione, che contiene gli elementi effettivi (cioè il codice) della query, ma non il suo risultato. Gli alberi di espressione rendono la struttura di un'espressione lambda trasparente ed esplicita.

Sintassi

- Espressione <TDelegate> name = lambdaExpression;

Parametri

Parametro	Dettagli
TDelegate	Il tipo di delegato da utilizzare per l'espressione
lambdaExpression	L'espressione lambda (es. `num => num < 5`)

Osservazioni

Introduzione agli alberi delle espressioni

Da dove veniamo

Gli alberi delle espressioni si occupano solo del consumo di "codice sorgente" in fase di esecuzione. Considerare un metodo che calcola l'imposta sulle vendite dovuta su un `decimal CalculateTotalTaxDue(SalesOrder order)` dell'ordine di vendita `decimal CalculateTotalTaxDue(SalesOrder order)` . L'utilizzo di tale metodo in un programma .NET è semplice: basta chiamarlo `decimal taxDue = CalculateTotalTaxDue(order);` . Cosa succede se si desidera applicarlo a tutti i risultati di una query remota (SQL, XML, un server remoto, ecc.)? Quelle fonti di query remote non possono chiamare il metodo! Tradizionalmente, dovresti invertire il flusso in tutti questi casi. Crea l'intera query, memorizzala in memoria, quindi visualizza i risultati e calcola le imposte per ogni risultato.

Come evitare i problemi di memoria e latenza di inversione del flusso

Gli alberi di espressione sono strutture di dati in un formato di un albero, in cui ogni nodo contiene un'espressione. Sono utilizzati per tradurre le istruzioni compilate (come i metodi utilizzati per filtrare i dati) in espressioni che potrebbero essere utilizzate al di fuori dell'ambiente del programma, come all'interno di una query del database.

Il problema qui è che una query remota *non può accedere al nostro metodo* . Potremmo evitare questo problema se, invece, abbiamo inviato le *istruzioni* per il metodo alla query remota. Nel nostro esempio `CalculateTotalTaxDue` , ciò significa che inviamo queste informazioni:

1. Creare una variabile per memorizzare l'imposta totale
2. Passa attraverso tutte le linee sull'ordine
3. Per ogni riga, controlla se il prodotto è tassabile
4. Se lo è, moltiplica la linea totale per l'aliquota fiscale applicabile e aggiungi tale importo al totale
5. Altrimenti non fare nulla

Con queste istruzioni, la query remota può eseguire il lavoro mentre crea i dati.

Ci sono due sfide per l'implementazione di questo. Come si trasforma un metodo .NET compilato in un elenco di istruzioni e come si formattano le istruzioni in modo che possano essere utilizzate dal sistema remoto?

Senza alberi di espressione, è possibile risolvere solo il primo problema con MSIL. (MSIL è il codice simile all'assembler creato dal compilatore .NET). L'analisi di MSIL è *possibile* , ma non è facile. Anche quando lo si analizza correttamente, può essere difficile determinare quale fosse l'intento del programmatore originale con una particolare routine.

Gli alberi delle espressioni salvano il giorno

Gli alberi delle espressioni affrontano questi problemi esatti. Rappresentano le istruzioni del programma una struttura di dati dell'albero in cui ogni nodo rappresenta *un'istruzione* e contiene riferimenti a tutte le informazioni necessarie per eseguire tale istruzione. Ad esempio, un `MethodCallExpression` ha riferimento a 1) il `MethodInfo` che chiamerà, 2) un elenco di `Expression` che passerà a quel metodo, 3) per i metodi di istanza, l' `Expression` cui chiamerai il metodo. Puoi "camminare sull'albero" e applicare le istruzioni sulla tua query remota.

Creazione di alberi di espressione

Il modo più semplice per creare un albero di espressioni è con un'espressione lambda. Queste espressioni sembrano quasi le stesse dei normali metodi C #. È importante rendersi conto che questo è *magia del compilatore* . Quando crei per la prima volta un'espressione lambda, il compilatore verifica a cosa lo assegni. Se si tratta di un tipo di `Delegate` (incluso `Action` o `Func`), il compilatore converte l'espressione lambda in un delegato. Se è una `LambdaExpression` (o `Expression<Action<T>>` o `Expression<Func<T>>` che sono fortemente digitate `LambdaExpression`), il compilatore la trasforma in `LambdaExpression` . Qui è dove entra la magia. Dietro le quinte, il compilatore *usa l'albero delle espressioni API* per trasformare la tua espressione lambda in una `LambdaExpression` .

Le espressioni Lambda non possono creare ogni tipo di albero delle espressioni. In questi casi, è possibile utilizzare manualmente l'API di espressioni per creare l'albero necessario. Nell'esempio Comprendere le espressioni API , creiamo l'espressione `CalculateTotalSalesTax` utilizzando l'API.

NOTA: i nomi diventano un po 'confusi qui. *Un'espressione lambda* (due parole, in minuscolo) si riferisce al blocco di codice con un indicatore `=>` . Rappresenta un metodo anonimo in C # e viene convertito in un `Delegate` o in `Expression` . A *`LambdaExpression`* (una parola, PascalCase) fa riferimento al tipo di nodo all'interno dell'API di espressione che rappresenta un metodo che è possibile eseguire.

Alberi di espressione e LINQ

Uno degli usi più comuni degli alberi di espressione è con LINQ e query di database. LINQ accoppia un albero di espressioni con un provider di query per applicare le istruzioni alla query remota di destinazione. Ad esempio, il provider di query LINQ to Entity Framework trasforma un albero di espressioni in SQL che viene eseguito direttamente sul database.

Mettendo insieme tutti i pezzi, puoi vedere il vero potere dietro LINQ.

1. Scrivi una query usando un'espressione lambda: `products.Where(x => x.Cost > 5)`
2. Il compilatore trasforma quell'espressione in un albero di espressioni con le istruzioni "controlla se la proprietà Cost del parametro è maggiore di cinque".
3. Il provider di query analizza la struttura dell'espressione e produce una query SQL valida `SELECT * FROM products WHERE Cost > 5`
4. L'ORM proietta tutti i risultati in POCO e ottieni un elenco di oggetti

Gli appunti

- Gli alberi delle espressioni sono immutabili. Se si desidera modificare un albero di espressioni è necessario crearne uno nuovo, copiare quello esistente in quello nuovo (per attraversare un albero di espressioni è possibile utilizzare `ExpressionVisitor`) e apportare le modifiche desiderate.

Examples

Creazione di alberi di espressione mediante l'API

```
using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
```

```
        new ParameterExpression[] { numParam });
```

Compilazione degli alberi di espressione

```
// Define an expression tree, taking an integer, returning a bool.
Expression<Func<int, bool>> expr = num => num < 5;

// Call the Compile method on the expression tree to return a delegate that can be called.
Func<int, bool> result = expr.Compile();

// Invoke the delegate and write the result to the console.
Console.WriteLine(result(4)); // Prints true

// Prints True.

// You can also combine the compile step with the call/invoke step as below:
Console.WriteLine(expr.Compile()(4));
```

Parsing Expression Trees

```
using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

// Decomposed expression: num => num LessThan 5
```

Crea alberi espressione con un'espressione lambda

Di seguito è riportato l'albero delle espressioni di base creato da lambda.

```
Expression<Func<int, bool>> lambda = num => num == 42;
```

Per creare alberi di espressioni 'a mano', si dovrebbe usare la classe `Expression` .

L'espressione sopra sarebbe equivalente a:

```
ParameterExpression parameter = Expression.Parameter(typeof(int), "num"); // num argument
ConstantExpression constant = Expression.Constant(42, typeof(int)); // 42 constant
BinaryExpression equality = Expression.Equals(parameter, constant); // equality of two
expressions (num == 42)
Expression<Func<int, bool>> lambda = Expression.Lambda<Func<int, bool>>(equality, parameter);
```

Comprendere l'API delle espressioni

Utilizzeremo l'albero delle espressioni API per creare un albero `CalculateSalesTax` . In inglese semplice, ecco un riepilogo dei passaggi necessari per creare l'albero.

1. Controlla se il prodotto è tassabile
2. Se lo è, moltiplica la linea totale per l'aliquota fiscale applicabile e restituisci tale importo
3. Altrimenti restituisci 0

```
//For reference, we're using the API to build this lambda expression
    orderLine => orderLine.IsTaxable ? orderLine.Total * orderLine.Order.TaxRate : 0;

//The orderLine parameter we pass in to the method.  We specify it's type (OrderLine) and the
name of the parameter.
    ParameterExpression orderLine = Expression.Parameter(typeof(OrderLine), "orderLine");

//Check if the parameter is taxable;  First we need to access the is taxable property, then
check if it's true
    PropertyInfo isTaxableAccessor = typeof(OrderLine).GetProperty("IsTaxable");
    MemberExpression getIsTaxable = Expression.MakeMemberAccess(orderLine, isTaxableAccessor);
    UnaryExpression isLineTaxable = Expression.IsTrue(getIsTaxable);

//Before creating the if, we need to create the braches
    //If the line is taxable, we'll return the total times the tax rate; get the total and tax
rate, then multiply
    //Get the total
    PropertyInfo totalAccessor = typeof(OrderLine).GetProperty("Total");
    MemberExpression getTotal = Expression.MakeMemberAccess(orderLine, totalAccessor);

    //Get the order
    PropertyInfo orderAccessor = typeof(OrderLine).GetProperty("Order");
    MemberExpression getOrder = Expression.MakeMemberAccess(orderLine, orderAccessor);

    //Get the tax rate - notice that we pass the getOrder expression directly to the member
access
    PropertyInfo taxRateAccessor = typeof(Order).GetProperty("TaxRate");
    MemberExpression getTaxRate = Expression.MakeMemberAccess(getOrder, taxRateAccessor);

    //Multiply the two - notice we pass the two operand expressions directly to multiply
    BinaryExpression multiplyTotalByRate = Expression.Multiply(getTotal, getTaxRate);

//If the line is not taxable, we'll return a constant value - 0.0 (decimal)
    ConstantExpression zero = Expression.Constant(0M);

//Create the actual if check and branches
    ConditionalExpression ifTaxableTernary = Expression.Condition(isLineTaxable,
multiplyTotalByRate, zero);

//Wrap the whole thing up in a "method" - a LambdaExpression
    Expression<Func<OrderLine, decimal>> method = Expression.Lambda<Func<OrderLine,
decimal>>(ifTaxableTernary, orderLine);
```

Expression Tree Basic

Gli alberi di espressione rappresentano il codice in una struttura dati ad albero, in cui ogni nodo è un'espressione

Expression Trees consente la modifica dinamica del codice eseguibile, l'esecuzione di query LINQ in vari database e la creazione di query dinamiche. È possibile compilare ed eseguire codice rappresentato da alberi di espressione.

Questi vengono anche utilizzati nel linguaggio dinamico run-time (DLR) per fornire l'interoperabilità tra i linguaggi dinamici e .NET Framework e per consentire ai produttori di compilatori di emettere alberi di espressioni anziché il linguaggio intermedio Microsoft (MSIL).

Gli alberi di espressione possono essere creati tramite

1. Espressione lambda anonima,
2. Manualmente utilizzando lo spazio dei nomi System.Linq.Expressions.

Expression Trees from Lambda Expressions

Quando un'espressione lambda viene assegnata alla variabile di tipo Expression, il compilatore emette il codice per creare un albero di espressioni che rappresenti l'espressione lambda.

I seguenti esempi di codice mostrano come fare in modo che il compilatore C # crei un albero di espressioni che rappresenti l'espressione lambda num => num <5.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Alberi di espressione utilizzando l'API

Alberi di espressione creati anche usando la classe di **espressione** . Questa classe contiene metodi factory statici che creano nodi di albero dell'espressione di tipi specifici.

Di seguito sono riportati alcuni tipi di nodi Tree.

1. ParameterExpression
2. MethodCallExpression

Il seguente esempio di codice mostra come creare un albero di espressioni che rappresenti l'espressione lambda num => num <5 utilizzando l'API.

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 = Expression.Lambda<Func<int, bool>>(numLessThanFive,new
ParameterExpression[] { numParam });
```

Esaminando la struttura di un'espressione usando Visitor

Definisci una nuova classe di visitatori sovrascrivendo alcuni dei metodi di ExpressionVisitor :

```
class PrintingVisitor : ExpressionVisitor {
    protected override Expression VisitConstant(ConstantExpression node) {
        Console.WriteLine("Constant: {0}", node);
        return base.VisitConstant(node);
```

```
    }
    protected override Expression VisitParameter(ParameterExpression node) {
        Console.WriteLine("Parameter: {0}", node);
        return base.VisitParameter(node);
    }
    protected override Expression VisitBinary(BinaryExpression node) {
        Console.WriteLine("Binary with operator {0}", node.NodeType);
        return base.VisitBinary(node);
    }
}
```

Chiama `Visit` per utilizzare questo visitatore su un'espressione esistente:

```
Expression<Func<int,bool>> isBig = a => a > 1000000;
var visitor = new PrintingVisitor();
visitor.Visit(isBig);
```

Leggi Alberi di espressione online: https://riptutorial.com/it/csharp/topic/75/alberi-di-espressione

Capitolo 6: Alias di tipi predefiniti

Examples

Tabella dei tipi incorporati

La tabella seguente mostra le parole chiave per i tipi di `C#` incorporati, che sono alias di tipi predefiniti negli spazi dei nomi di sistema.

C # Type	Tipo di .NET Framework
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
carbonizzare	System.Char
decimale	System.Decimal
Doppio	System.Double
galleggiante	System.Single
int	System.Int32
uint	System.UInt32
lungo	System.Int64
ulong	System.UInt64
oggetto	System.Object
corto	System.Int16
USHORT	System.UInt16
stringa	System.String

Le parole chiave di tipo `C#` e i loro alias sono intercambiabili. Ad esempio, è possibile dichiarare una variabile intera utilizzando una delle seguenti dichiarazioni:

```
int number = 123;
System.Int32 number = 123;
```

Leggi Alias di tipi predefiniti online: https://riptutorial.com/it/csharp/topic/1862/alias---di-tipi-predefiniti

Capitolo 7: Annotazione dei dati

Examples

DisplayNameAttribute (attributo di visualizzazione)

`DisplayName` imposta il nome visualizzato per un metodo property, event o public void con argomenti zero (0).

```
public class Employee
{
    [DisplayName(@"Employee first name")]
    public string FirstName { get; set; }
}
```

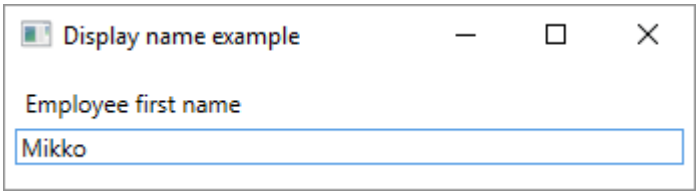
Semplice esempio di utilizzo nell'applicazione XAML

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:wpfApplication="clr-namespace:WpfApplication"
        Height="100" Width="360" Title="Display name example">

    <Window.Resources>
        <wpfApplication:DisplayNameConverter x:Key="DisplayNameConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- Label (DisplayName attribute) -->
        <Label Content="{Binding Employee, Converter={StaticResource DisplayNameConverter},
ConverterParameter=FirstName}" />
        <!-- TextBox (FirstName property value) -->
        <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}" />
    </StackPanel>

</Window>
```

```
namespace WpfApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee();

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }
```

```
        public Employee Employee
        {
            get { return _employee; }
            set { _employee = value; }
        }
    }
}
```

```
namespace WpfApplication
{
    public class DisplayNameConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            // Get display name for given instance type and property name
            var attribute = value.GetType()
                .GetProperty(parameter.ToString())
                .GetCustomAttributes(false)
                .OfType<DisplayNameAttribute>()
                .FirstOrDefault();

            return attribute != null ? attribute.DisplayName : string.Empty;
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            throw new NotImplementedException();
        }
    }
}
```



EditableAttribute (attributo di modellazione dati)

`EditableAttribute` stabilisce se gli utenti devono essere in grado di modificare il valore della proprietà della classe.

```
public class Employee
{
    [Editable(false)]
    public string FirstName { get; set; }
}
```

Semplice esempio di utilizzo nell'applicazione XAML

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:wpfApplication="clr-namespace:WpfApplication"
        Height="70" Width="360" Title="Display name example">

    <Window.Resources>
        <wpfApplication:EditableConverter x:Key="EditableConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- TextBox Text (FirstName property value) -->
        <!-- TextBox IsEnabled (Editable attribute) -->
        <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"
                 IsEnabled="{Binding Employee, Converter={StaticResource EditableConverter},
ConverterParameter=FirstName}"/>
    </StackPanel>

</Window>
```

```
namespace WpfApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee() { FirstName = "This is not editable"};

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }

        public Employee Employee
        {
            get { return _employee; }
            set { _employee = value; }
        }
    }
}
```

```
namespace WpfApplication
{
    public class EditableConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            // return editable attribute's value for given instance property,
            // defaults to true if not found
            var attribute = value.GetType()
                .GetProperty(parameter.ToString())
                .GetCustomAttributes(false)
                .OfType<EditableAttribute>()
                .FirstOrDefault();

            return attribute != null ? attribute.AllowEdit : true;
```

```
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

Attributi di convalida

Gli attributi di convalida vengono utilizzati per applicare varie regole di convalida in modo dichiarativo su classi o membri della classe. Tutti gli attributi di convalida derivano dalla classe base ValidationAttribute .

Esempio: RequiredAttribute

Se convalidato tramite il metodo `ValidationAttribute.Validate` , questo attributo restituirà un errore se la proprietà `Name` è null o contiene solo spazi.

```
public class ContactModel
{
    [Required(ErrorMessage = "Please provide a name.")]
    public string Name { get; set; }
}
```

Esempio: StringLengthAttribute

`StringLengthAttribute` convalida se una stringa è inferiore alla lunghezza massima di una stringa. Può facoltativamente specificare una lunghezza minima. Entrambi i valori sono inclusi.

```
public class ContactModel
{
    [StringLength(20, MinimumLength = 5, ErrorMessage = "A name must be between five and
twenty characters.")]
    public string Name { get; set; }
}
```

Esempio: RangeAttribute

`RangeAttribute` fornisce il valore massimo e minimo per un campo numerico.

```
public class Model
{
    [Range(0.01, 100.00,ErrorMessage = "Price must be between 0.01 and 100.00")]
    public decimal Price { get; set; }
}
```

Esempio: CustomValidationAttribute

La classe `CustomValidationAttribute` consente di richiamare un metodo `static` personalizzato per la convalida. Il metodo personalizzato deve essere `static ValidationResult [MethodName] (object input)` .

```
public class Model
{
    [CustomValidation(typeof(MyCustomValidation), "IsNotAnApple")]
    public string FavoriteFruit { get; set; }
}
```

Dichiarazione del metodo:

```
public static class MyCustomValidation
{
    public static ValidationResult IsNotAnApple(object input)
    {
        var result = ValidationResult.Success;

        if (input?.ToString()?.ToUpperInvariant() == "APPLE")
        {
            result = new ValidationResult("Apples are not allowed.");
        }

        return result;
    }
}
```

Creazione di un attributo di convalida personalizzato

Gli attributi di convalida personalizzati possono essere creati derivando dalla classe di base `ValidationAttribute` , quindi sovrascrivendo `virtual` metodi `virtual` secondo necessità.

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = false)]
public class NotABananaAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        var inputValue = value as string;
        var isValid = true;

        if (!string.IsNullOrEmpty(inputValue))
        {
            isValid = inputValue.ToUpperInvariant() != "BANANA";
        }
```

```
        return isValid;
    }
}
```

Questo attributo può quindi essere usato in questo modo:

```
public class Model
{
    [NotABanana(ErrorMessage = "Bananas are not allowed.")]
    public string FavoriteFruit { get; set; }
}
```

Nozioni di base sull'annotazione dei dati

Le annotazioni dei dati sono un modo per aggiungere più informazioni contestuali alle classi o ai membri di una classe. Esistono tre categorie principali di annotazioni:

- Attributi di convalida: aggiungi criteri di convalida ai dati
- Visualizza attributi: specifica come i dati devono essere visualizzati all'utente
- Attributi di modellazione: aggiungi informazioni sull'utilizzo e sulla relazione con altre classi

uso

Ecco un esempio in cui vengono utilizzati due `ValidationAttribute` e uno `DisplayAttribute` :

```
class Kid
{
    [Range(0, 18)] // The age cannot be over 18 and cannot be negative
    public int Age { get; set; }
    [StringLength(MaximumLength = 50, MinimumLength = 3)] // The name cannot be under 3 chars
or more than 50 chars
    public string Name { get; set; }
    [DataType(DataType.Date)] // The birthday will be displayed as a date only (without the
time)
    public DateTime Birthday { get; set; }
}
```

Le annotazioni dei dati sono utilizzate principalmente in framework come ASP.NET. Ad esempio, in `ASP.NET MVC` , quando un modello viene ricevuto da un metodo controller, `ModelState.IsValid()` può essere utilizzato per determinare se il modello ricevuto rispetta tutto il suo `ValidationAttribute` . `DisplayAttribute` viene anche utilizzato in `ASP.NET MVC` per determinare come visualizzare i valori su una pagina Web.

Esegui manualmente gli attributi di convalida

La maggior parte delle volte, gli attributi di convalida sono utilizzati all'interno di framework (come ASP.NET). Questi framework si occupano dell'esecuzione degli attributi di convalida. Ma cosa succede se si desidera eseguire manualmente gli attributi di convalida? Basta usare la classe `Validator` (nessuna riflessione necessaria).

Contesto di validazione

Qualsiasi convalida necessita di un contesto per fornire alcune informazioni su ciò che viene convalidato. Questo può includere varie informazioni come l'oggetto da convalidare, alcune proprietà, il nome da visualizzare nel messaggio di errore, ecc.

```
ValidationContext vc = new ValidationContext(objectToValidate); // The simplest form of
validation context. It contains only a reference to the object being validated.
```

Una volta creato il contesto, ci sono molti modi per fare la convalida.

Convalidare un oggetto e tutte le sue proprietà

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the
results of the validation
bool isValid = Validator.TryValidateObject(objectToValidate, vc, results, true); // Validates
the object and its properties using the previously created context.
// The variable isValid will be true if everything is valid
// The results variable contains the results of the validation
```

Convalidare una proprietà di un oggetto

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the
results of the validation
bool isValid = Validator.TryValidatePropery(objectToValidate.PropertyToValidate, vc, results,
true); // Validates the property using the previously created context.
// The variable isValid will be true if everything is valid
// The results variable contains the results of the validation
```

E altro ancora

Per saperne di più sulla convalida manuale vedi:

- Documentazione di classe ValidationContext
- Documentazione della classe Validator

Leggi Annotazione dei dati online: https://riptutorial.com/it/csharp/topic/4942/annotazione-dei-dati

Capitolo 8: Argomenti nominati

Examples

Gli argomenti con nome possono rendere il tuo codice più chiaro

Considera questa semplice classe:

```
class SmsUtil
{
    public bool SendMessage(string from, string to, string message, int retryCount, object
attachment)
    {
         // Some code
    }
}
```

Prima del C # 3.0 era:

```
var result = SmsUtil.SendMessage("Mehran", "Maryam", "Hello there!", 12, null);
```

puoi rendere questo metodo ancora più chiaro con gli **argomenti** con **nome** :

```
var result = SmsUtil.SendMessage(
    from: "Mehran",
    to:  "Maryam",
    message "Hello there!",
    retryCount: 12,
    attachment: null);
```

Argomenti con nome e parametri opzionali

È possibile combinare argomenti con nome con parametri opzionali.

Vediamo questo metodo:

```
public sealed class SmsUtil
{
    public static bool SendMessage(string from, string to, string message, int retryCount = 5,
object attachment = null)
    {
         // Some code
    }
}
```

Quando si desidera chiamare questo metodo *senza* impostare `retryCount` argomento `retryCount` :

```
var result = SmsUtil.SendMessage(
                        from       : "Cihan",
                        to         : "Yakar",
```

```
                        message    : "Hello there!",
                        attachment : new object());
```

L'ordine degli argomenti non è necessario

Puoi inserire argomenti con nome nell'ordine che preferisci.

Metodo di esempio:

```
public static string Sample(string left, string right)
{
     return string.Join("-",left,right);
}
```

Chiama il campione:

```
Console.WriteLine (Sample(left:"A",right:"B"));
Console.WriteLine (Sample(right:"A",left:"B"));
```

risultati:

```
A-B
B-A
```

Argomenti con nome evita bug sui parametri opzionali

Usa sempre gli argomenti con nome sui parametri facoltativi, per evitare potenziali bug quando il metodo viene modificato.

```
class Employee
{
    public string Name { get; private set; }

    public string Title { get; set; }

    public Employee(string name = "<No Name>", string title = "<No Title>")
    {
        this.Name = name;
        this.Title = title;
    }
}

var jack = new Employee("Jack", "Associate");   //bad practice in this line
```

Il codice sopra compila e funziona bene, finché il costruttore non viene cambiato un giorno come:

```
//Evil Code: add optional parameters between existing optional parameters
public Employee(string name = "<No Name>", string department = "intern", string title = "<No
Title>")
{
    this.Name = name;
    this.Department = department;
```

```
    this.Title = title;
}

//the below code still compiles, but now "Associate" is an argument of "department"
var jack = new Employee("Jack", "Associate");
```

Le migliori pratiche per evitare errori quando "qualcun altro nel team" ha commesso degli errori:

```
var jack = new Employee(name: "Jack", title: "Associate");
```

Leggi Argomenti nominati online: https://riptutorial.com/it/csharp/topic/2076/argomenti-nominati

Capitolo 9: Argomenti nominati e opzionali

Osservazioni

Argomenti nominati

Rif: MSDN Gli argomenti con nome consentono di specificare un argomento per un particolare parametro associando l'argomento al nome del parametro anziché con la posizione del parametro nell'elenco dei parametri.

Come detto da MSDN, un argomento con nome,

- Consente di passare l'argomento alla funzione associando il nome del parametro.
- Non c'è bisogno di ricordare la posizione dei parametri che non siamo a conoscenza di sempre.
- Non c'è bisogno di guardare l'ordine dei parametri nella lista dei parametri della funzione chiamata.
- Possiamo specificare il parametro per ogni argomento in base al suo nome.

Argomenti opzionali

Rif: MSDN La definizione di un metodo, costruttore, indicizzatore o delegato può specificare che i suoi parametri sono obbligatori o che sono facoltativi. Ogni chiamata deve fornire argomenti per tutti i parametri richiesti, ma può omettere argomenti per i parametri facoltativi.

Come detto da MSDN, un argomento facoltativo,

- Possiamo omettere l'argomento nella chiamata se tale argomento è un argomento facoltativo
- Ogni argomento facoltativo ha il proprio valore predefinito
- Prenderà il valore predefinito se non forniamo il valore
- Un valore predefinito di un argomento facoltativo deve essere a
    - Espressione costante
    - Deve essere un tipo di valore come enum o struct.
    - Deve essere un'espressione di default del modulo (valueType)
- Deve essere impostato alla fine dell'elenco dei parametri

Examples

Argomenti nominati

Considera che seguire è la nostra chiamata di funzione.

```
FindArea(120, 56);
```

In questo il nostro primo argomento è la lunghezza (cioè 120) e il secondo argomento è larghezza

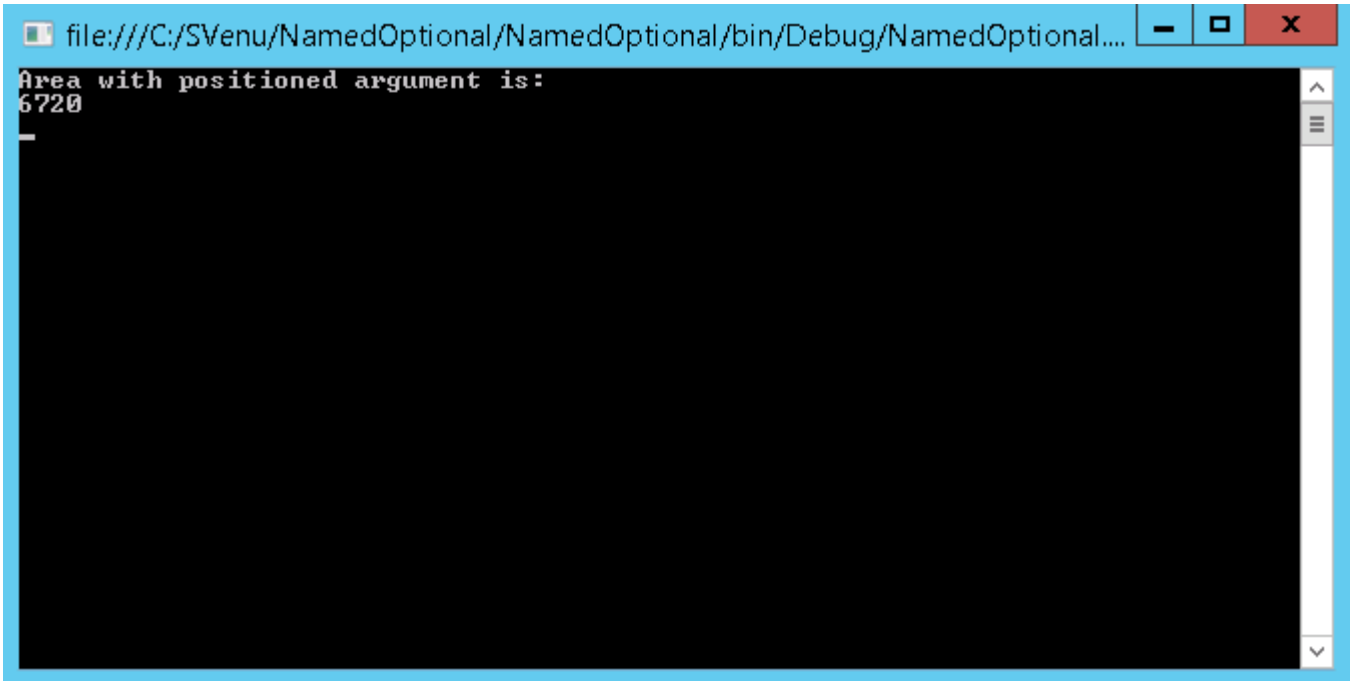
(cioè 56). E stiamo calcolando l'area con quella funzione. E di seguito è la definizione della funzione.

```
private static double FindArea(int length, int width)
       {
           try
           {
               return (length* width);
           }
           catch (Exception)
           {
               throw new NotImplementedException();
           }
       }
```

Quindi nella prima chiamata di funzione, abbiamo appena passato gli argomenti per la sua posizione. Destra?

```
double area;
Console.WriteLine("Area with positioned argument is: ");
area = FindArea(120, 56);
Console.WriteLine(area);
Console.Read();
```

Se lo esegui, otterrai un output come segue.



Ora ecco le caratteristiche di un argomento con nome. Si prega di consultare la chiamata alla funzione precedente.

```
Console.WriteLine("Area with Named argument is: ");
area = FindArea(length: 120, width: 56);
Console.WriteLine(area);
Console.Read();
```

Qui stiamo dando gli argomenti nominati nella chiamata al metodo.

```
area = FindArea(length: 120, width: 56);
```

Ora se si esegue questo programma, si otterrà lo stesso risultato. Possiamo dare i nomi vice versa nella chiamata al metodo se stiamo usando gli argomenti con nome.

```
Console.WriteLine("Area with Named argument vice versa is: ");
area = FindArea(width: 120, length: 56);
Console.WriteLine(area);
Console.Read();
```

Uno degli usi importanti di un argomento con nome è che quando lo usi nel tuo programma migliora la leggibilità del tuo codice. Dice semplicemente quale dovrebbe essere la tua argomentazione o cosa è?

Puoi anche dare gli argomenti posizionali. Ciò significa, una combinazione di entrambi gli argomenti posizionali e argomento con nome.

```
Console.WriteLine("Area with Named argument Positional Argument : ");
            area = FindArea(120, width: 56);
            Console.WriteLine(area);
            Console.Read();
```

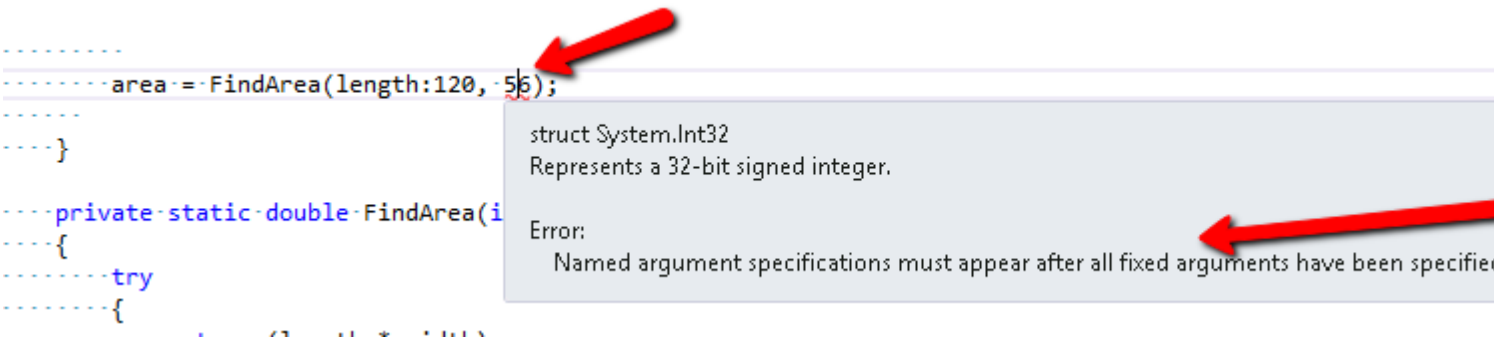
Nell'esempio precedente abbiamo passato 120 come lunghezza e 56 come argomento con nome per la larghezza del parametro.

Ci sono anche alcune limitazioni. Discuteremo la limitazione di un argomento chiamato ora.

Limitazione dell'uso di un argomento con nome

La specifica dell'argomento con nome deve apparire dopo che sono stati specificati tutti gli argomenti fissi.

Se si usa un argomento con nome prima di un argomento fisso, si otterrà un errore in fase di compilazione come segue.



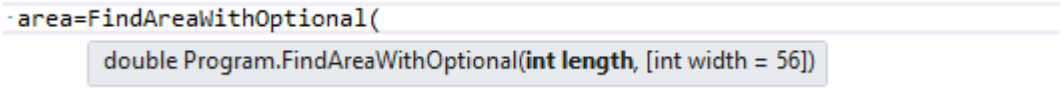
La specifica dell'argomento con nome deve apparire dopo che sono stati specificati tutti gli argomenti fissi

Argomenti opzionali

Considerare precedente è la nostra definizione di funzione con argomenti opzionali.

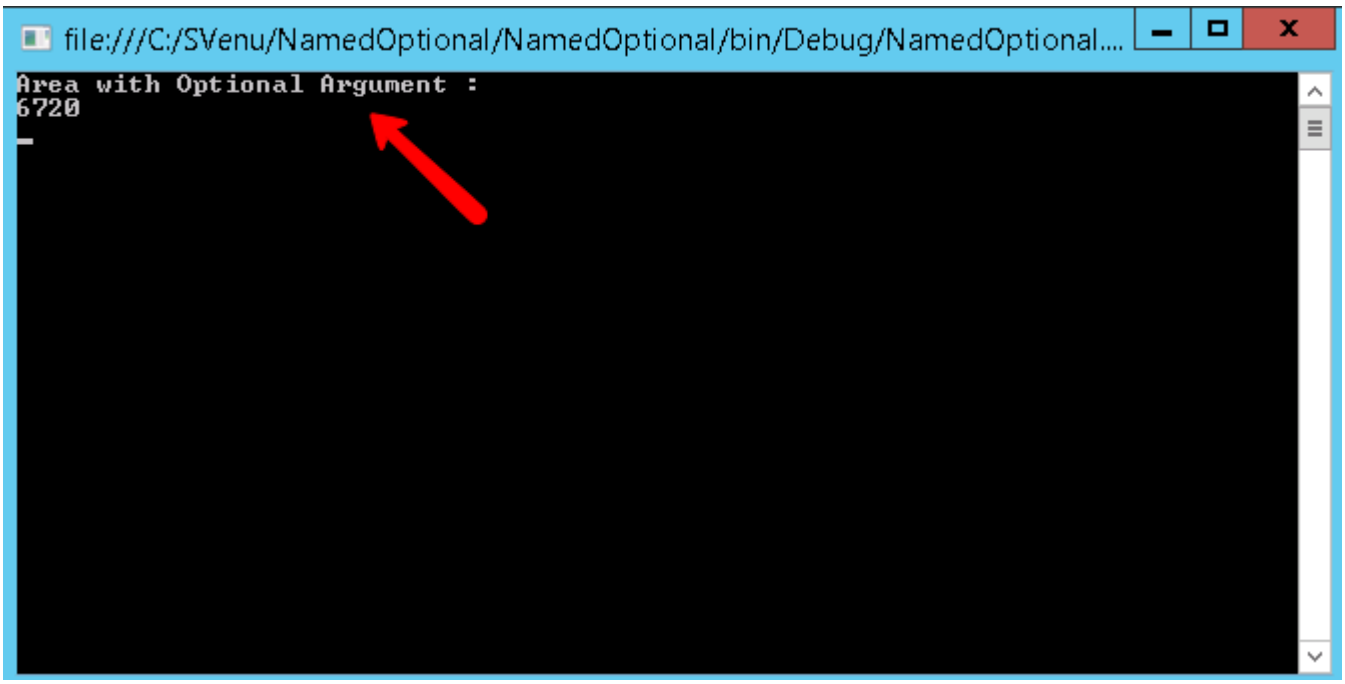
```
private static double FindAreaWithOptional(int length, int width=56)
       {
           try
           {
               return (length * width);
           }
           catch (Exception)
           {
               throw new NotImplementedException();
           }
       }
```

Qui abbiamo impostato il valore per width come facoltativo e abbiamo dato valore come 56. Se si nota, lo stesso IntelliSense mostra l'argomento opzionale come mostrato nell'immagine sottostante.




```
Console.WriteLine("Area with Optional Argument : ");
area = FindAreaWithOptional(120);
Console.WriteLine(area);
Console.Read();
```

Nota che non abbiamo ricevuto alcun errore durante la compilazione e ti forniremo un output come segue.



Utilizzando l'attributo opzionale.

Un altro modo per implementare l'argomento facoltativo è utilizzare la parola chiave `[Optional]` . Se non si passa il valore per l'argomento facoltativo, il valore predefinito di tale tipo di dati viene assegnato a tale argomento. La parola chiave `Optional` è presente nello spazio dei nomi "Runtime.InteropServices".

```
using System.Runtime.InteropServices;
private static double FindAreaWithOptional(int length, [Optional]int width)
   {
       try
       {
           return (length * width);
       }
       catch (Exception)
       {
           throw new NotImplementedException();
       }
   }

area = FindAreaWithOptional(120);  //area=0
```

E quando chiamiamo la funzione, otteniamo 0 perché il secondo argomento non viene passato e il valore predefinito di int è 0 e quindi il prodotto è 0.

Leggi Argomenti nominati e opzionali online: https://riptutorial.com/it/csharp/topic/5220/argomenti-nominati-e-opzionali

Capitolo 10: Array

Sintassi

- **Dichiarazione di un array:**

  <tipo> [] <nome>;

- **Dichiarazione di array bidimensionale:**

  <tipo> [,] <nome> = nuovo <tipo> [<valore>, <valore>];

- **Dichiarazione di una matrice seghettata:**

  <tipo> [] <nome> = nuovo <tipo> [<valore>];

- **Dichiarazione di un sottoarray per una matrice frastagliata:**

  <nome> [<valore>] = nuovo <tipo> [<valore>];

- **Inizializzazione di un array senza valori:**

  <nome> = nuovo <tipo> [<lunghezza>];

- **Inizializzazione di un array con valori:**

  <nome> = nuovo <tipo> [] {<valore>, <valore>, <valore>, ...};

- **Inizializzazione di un array bidimensionale con valori:**

  <nome> = nuovo <tipo> [,] {{valore>, <valore>}, {<valore>, <valore>}, ...};

- **Accedere a un elemento all'indice i:**

  <Nome> [i]

- **Ottenere la lunghezza dell'array:**

  <Nome> .Length

Osservazioni

In C #, un array è un tipo di riferimento, il che significa che è *nullable* .

Un array ha una lunghezza fissa, il che significa che non puoi `.Add()` ad esso o `.Remove()` da esso. Per poterli utilizzare, è necessario un array dinamico: `List` o `ArrayList` .

Examples

Covarianza di matrice

```
string[] strings = new[] {"foo", "bar"};
object[] objects = strings; // implicit conversion from string[] to object[]
```

Questa conversione non è sicura dal testo. Il seguente codice genererà un'eccezione di runtime:

```
string[] strings = new[] {"Foo"};
object[] objects = strings;

objects[0] = new object(); // runtime exception, object is not string
string str = strings[0];   // would have been bad if above assignment had succeeded
```

Ottenere e impostare i valori dell'array

```
int[] arr = new int[] { 0, 10, 20, 30};

// Get
Console.WriteLine(arr[2]); // 20

// Set
arr[2] = 100;

// Get the updated value
Console.WriteLine(arr[2]); // 100
```

Dichiarare un array

Un array può essere dichiarato e riempito con il valore predefinito usando la sintassi di inizializzazione parentesi quadra (`[]`). Ad esempio, creando un array di 10 numeri interi:

```
int[] arr = new int[10];
```

Gli indici in C # sono a base zero. Gli indici dell'array sopra saranno 0-9. Per esempio:

```
int[] arr = new int[3] {7,9,4};
Console.WriteLine(arr[0]); //outputs 7
Console.WriteLine(arr[1]); //outputs 9
```

Il che significa che il sistema inizia a contare l'indice degli elementi da 0. Inoltre, gli accessi agli elementi degli array vengono eseguiti in **un tempo costante** . Ciò significa che l'accesso al primo elemento dell'array ha lo stesso costo (nel tempo) dell'accesso al secondo elemento, al terzo elemento e così via.

Si può anche dichiarare un riferimento nudo a un array senza istanziare un array.

```
int[] arr = null;   // OK, declares a null reference to an array.
int first = arr[0]; // Throws System.NullReferenceException because there is no actual array.
```

Un array può anche essere creato e inizializzato con valori personalizzati utilizzando la sintassi di

inizializzazione della raccolta:

```
int[] arr = new int[] { 24, 2, 13, 47, 45 };
```

La `new int[]` porzione `new int[]` può essere omessa quando si dichiara una variabile di array. Questa non è *un'espressione* autonoma, quindi utilizzarla come parte di una chiamata diversa non funziona (per questo, usa la versione con una `new`):

```
int[] arr = { 24, 2, 13, 47, 45 };  // OK
int[] arr1;
arr1 = { 24, 2, 13, 47, 45 };       // Won't compile
```

Array implicitamente tipizzati

In alternativa, in combinazione con la parola chiave `var` , il tipo specifico può essere omesso in modo da inferire il tipo della matrice:

```
// same as int[]
var arr = new [] { 1, 2, 3 };
// same as string[]
var arr = new [] { "one", "two", "three" };
// same as double[]
var arr = new [] { 1.0, 2.0, 3.0 };
```

Scorrere su un array

```
int[] arr = new int[] {1, 6, 3, 3, 9};

for (int i = 0; i < arr.Length; i++)
{
    Console.WriteLine(arr[i]);
}
```

utilizzando foreach:

```
foreach (int element in arr)
{
    Console.WriteLine(element);
}
```

utilizzando l'accesso non sicuro con i puntatori https://msdn.microsoft.com/en-ca/library/y31yhkeb.aspx

```
unsafe
{
    int length = arr.Length;
    fixed (int* p = arr)
    {
        int* pInt = p;
        while (length-- > 0)
        {
            Console.WriteLine(*pInt);
```

```
            pInt++;// move pointer to next element
        }
    }
}
```

Produzione:

1
6
3
3
9

Matrici multidimensionali

Le matrici possono avere più di una dimensione. Nell'esempio seguente viene creato un array bidimensionale di dieci righe e dieci colonne:

```
int[,] arr = new int[10, 10];
```

Un array di tre dimensioni:

```
int[,,] arr = new int[10, 10, 10];
```

È anche possibile inizializzare la matrice sulla dichiarazione:

```
int[,] arr = new int[4, 2] { {1, 1}, {2, 2}, {3, 3}, {4, 4} };

// Access a member of the multi-dimensional array:
Console.Out.WriteLine(arr[3, 1]);  // 4
```

Matrici frastagliate

Gli array frastagliati sono array che invece di tipi primitivi contengono array (o altre raccolte). È come un array di array: ogni elemento dell'array contiene un altro array.

Sono simili agli array multidimensionali, ma presentano una leggera differenza: poiché gli array multidimensionali sono limitati a un numero fisso di righe e colonne, con matrici frastagliate, ogni riga può avere un numero diverso di colonne.

Dichiarare una matrice seghettata

Ad esempio, dichiarando una matrice frastagliata con 8 colonne:

```
int[][] a = new int[8][];
```

Il secondo `[]` viene inizializzato senza un numero. Per inizializzare i sottoarray, è necessario farlo separatamente:

```
for (int i = 0; i < a.length; i++)
{
    a[i] = new int[10];
}
```

Ottenimento / impostazione dei valori

Ora, ottenere uno dei sottotitoli è facile. Stampiamo tutti i numeri della 3a colonna di `a` :

```
for (int i = 0; i < a[2].length; i++)
{
    Console.WriteLine(a[2][i]);
}
```

Ottenere un valore specifico:

```
a[<row_number>][<column_number>]
```

Impostazione di un valore specifico:

```
a[<row_number>][<column_number>] = <value>
```

Ricorda : è sempre consigliabile utilizzare matrici frastagliate (matrici di matrici) piuttosto che array multidimensionali (matrici). È più veloce e più sicuro da usare.

Nota sull'ordine delle parentesi

Considera una matrice tridimensionale di matrici a cinque dimensioni di matrici unidimensionali di `int` . Questo è scritto in C # come:

```
int[,,][,,,,][] arr = new int[8, 10, 12][,,,,][];
```

Nel sistema tipo CLR, la convenzione per l'ordinamento delle staffe è invertita, quindi con quanto sopra `arr` esempio abbiamo:

```
    arr.GetType().ToString() == "System.Int32[][,,,,][,,]"
```

e allo stesso modo:

```
    typeof(int[,,][,,,,][]).ToString() == "System.Int32[][,,,,][,,]"
```

Verifica se un array contiene un altro array

```
public static class ArrayHelpers
{
    public static bool Contains<T>(this T[] array, T[] candidate)
    {
        if (IsEmptyLocate(array, candidate))
```

```
            return false;

        if (candidate.Length > array.Length)
            return false;

        for (int a = 0; a <= array.Length - candidate.Length; a++)
        {
            if (array[a].Equals(candidate[0]))
            {
                int i = 0;
                for (; i < candidate.Length; i++)
                {
                    if (false == array[a + i].Equals(candidate[i]))
                        break;
                }
                if (i == candidate.Length)
                    return true;
            }
        }
        return false;
    }

    static bool IsEmptyLocate<T>(T[] array, T[] candidate)
    {
        return array == null
            || candidate == null
            || array.Length == 0
            || candidate.Length == 0
            || candidate.Length > array.Length;
    }
}
```

/// Campione

```
byte[] EndOfStream = Encoding.ASCII.GetBytes("---3141592---");
byte[] FakeReceivedFromStream = Encoding.ASCII.GetBytes("Hello, world!!!---3141592---");
if (FakeReceivedFromStream.Contains(EndOfStream))
{
    Console.WriteLine("Message received");
}
```

Inizializzazione di un array riempito con un valore non predefinito ripetuto

Come sappiamo, possiamo dichiarare un array con valori predefiniti:

```
int[] arr = new int[10];
```

Questo creerà una matrice di 10 numeri interi con ogni elemento dell'array che ha valore `0` (il valore predefinito di tipo `int`).

Per creare un array inizializzato con un valore non predefinito, è possibile utilizzare `Enumerable.Repeat` dallo `System.Linq` nomi `System.Linq` :

1. Per creare un array `bool` di dimensioni 10 pieno di **"true"**

```
bool[] booleanArray = Enumerable.Repeat(true, 10).ToArray();
```

2. Per creare un array `int` di dimensione 5 riempito con **"100"**

```
int[] intArray = Enumerable.Repeat(100, 5).ToArray();
```

3. Per creare una serie di `string` di dimensione 5 riempita con **"C #"**

```
string[] strArray = Enumerable.Repeat("C#", 5).ToArray();
```

Copia di array

Copia di una matrice parziale con il metodo statico `Array.Copy()` , a partire dall'indice 0 in entrambi, origine e destinazione:

```
var sourceArray = new int[] { 11, 12, 3, 5, 2, 9, 28, 17 };
var destinationArray= new int[3];
Array.Copy(sourceArray, destinationArray, 3);

// destinationArray will have 11,12 and 3
```

Copia l'intero array con il metodo di istanza `CopyTo()` , iniziando dall'indice 0 dell'origine e dall'indice specificato nella destinazione:

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = new int[6];
sourceArray.CopyTo(destinationArray, 2);

// destinationArray will have 0, 0, 11, 12, 7 and 0
```

`Clone` è usato per creare una copia di un oggetto array.

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = (int)sourceArray.Clone();

//destinationArray will be created and will have 11,12,17.
```

Sia `CopyTo` che `Clone` eseguono una copia superficiale che significa che il contenuto contiene riferimenti allo stesso oggetto degli elementi nell'array originale.

Creare una serie di numeri sequenziali

LINQ fornisce un metodo che semplifica la creazione di una raccolta piena di numeri sequenziali. Ad esempio, è possibile dichiarare una matrice che contiene gli interi tra 1 e 100.

Il metodo `Enumerable.Range` ci consente di creare una sequenza di numeri interi da una posizione iniziale specificata e un numero di elementi.

Il metodo accetta due argomenti: il valore iniziale e il numero di elementi da generare.

```
Enumerable.Range(int start, int count)
```

Si noti che il `count` non può essere negativo.

Uso:

```
int[] sequence = Enumerable.Range(1, 100).ToArray();
```

Questo genererà un array contenente i numeri da 1 a 100 (`[1, 2, 3, ..., 98, 99, 100]`).

Poiché il metodo `Range` restituisce un oggetto `IEnumerable<int>` , possiamo utilizzare altri metodi LINQ su di esso:

```
int[] squares = Enumerable.Range(2, 10).Select(x => x * x).ToArray();
```

Questo genererà una matrice che contiene 10 numeri interi a partire da `4` : `[4, 9, 16, ..., 100, 121]` .

Confronto tra array per l'uguaglianza

LINQ fornisce una funzione integrata per il controllo dell'uguaglianza di due `IEnumerable` , e tale funzione può essere utilizzata sugli array.

La funzione `SequenceEqual` restituirà `true` se gli array hanno la stessa lunghezza ei valori negli indici corrispondenti sono uguali e `false` altrimenti.

```
int[] arr1 = { 3, 5, 7 };
int[] arr2 = { 3, 5, 7 };
bool result = arr1.SequenceEqual(arr2);
Console.WriteLine("Arrays equal? {0}", result);
```

Questo stamperà:

```
Arrays equal? True
```

Array come istanze IEnumerable <>

Tutti gli array implementano l'interfaccia `IList` non generica (e quindi le interfacce di base `ICollection` e `IEnumerable` non generiche).

Ancora più importante, gli array unidimensionali implementano le interfacce generiche `IList<>` e `IReadOnlyList<>` (e le loro interfacce di base) per il tipo di dati che contengono. Ciò significa che possono essere trattati come tipi enumerabili generici e inoltrati a una varietà di metodi senza dover prima convertirli in un modulo non array.

```
int[] arr1 = { 3, 5, 7 };
IEnumerable<int> enumerableIntegers = arr1; //Allowed because arrays implement IEnumerable<T>
```

```
List<int> listOfIntegers = new List<int>();
listOfIntegers.AddRange(arr1); //You can pass in a reference to an array to populate a List.
```

Dopo aver eseguito questo codice, la lista `listOfIntegers` conterrà una `List<int>` contenente i valori 3, 5 e 7.

Il supporto `IEnumerable<>` significa che gli array possono essere interrogati con LINQ, ad esempio `arr1.Select(i => 10 * i)` .

Leggi Array online: https://riptutorial.com/it/csharp/topic/1429/array

Capitolo 11: AssemblyInfo.cs Esempi

Osservazioni

Il nome file `AssemblyInfo.cs` viene utilizzato per convenzione come il file di origine in cui gli sviluppatori inseriscono attributi di metadati che descrivono l'intero assembly che stanno creando.

Examples

[AssemblyTitle]

Questo attributo è usato per dare un nome a questo particolare assemblaggio.

```
[assembly: AssemblyTitle("MyProduct")]
```

[AssemblyProduct]

Questo attributo viene utilizzato per descrivere il prodotto per cui è destinato questo particolare assieme. Più assiemi possono essere componenti dello stesso prodotto, nel qual caso possono tutti condividere lo stesso valore per questo attributo.

```
[assembly: AssemblyProduct("MyProduct")]
```

AssemblyInfo globale e locale

Avere un globale consente una migliore DEUMIDITÀ, è necessario solo inserire valori diversi in AssemblyInfo.cs per i progetti con varianza. Questo uso presuppone che il tuo prodotto abbia più di un progetto di Visual Studio.

GlobalAssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.InteropServices;
//using Stackoverflow domain as a made up example

// It is common, and mostly good, to use one GlobalAssemblyInfo.cs that is added
// as a link to many projects of the same product, details below
// Change these attribute values in local assembly info to modify the information.
[assembly: AssemblyProduct("Stackoverflow Q&A")]
[assembly: AssemblyCompany("Stackoverflow")]
[assembly: AssemblyCopyright("Copyright © Stackoverflow 2016")]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("4e4f2d33-aaab-48ea-a63d-1f0a8e3c935f")]
[assembly: ComVisible(false)] //not going to expose ;)

// Version information for an assembly consists of the following four values:
// roughly translated from I reckon it is for SO, note that they most likely
```

```
// dynamically generate this file
//      Major Version  - Year 6 being 2016
//      Minor Version  - The month
//      Day Number     - Day of month
//      Revision       - Build number
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below: [assembly: AssemblyVersion("year.month.day.*")]
[assembly: AssemblyVersion("2016.7.00.00")]
[assembly: AssemblyFileVersion("2016.7.27.3839")]
```

AssemblyInfo.cs - uno per ogni progetto

```
//then the following might be put into a separate Assembly file per project, e.g.
[assembly: AssemblyTitle("Stackoveflow.Redis")]
```

È possibile aggiungere GlobalAssemblyInfo.cs al progetto locale utilizzando la seguente procedura :

1. Seleziona Aggiungi / elemento esistente ... nel menu di scelta rapida del progetto
2. Seleziona GlobalAssemblyInfo.cs
3. Espandi il pulsante Aggiungi facendo clic su quella piccola freccia in giù sulla mano destra
4. Seleziona "Aggiungi come collegamento" nell'elenco a discesa dei pulsanti

[AssemblyVersion]

Questo attributo applica una versione all'assembly.

```
[assembly: AssemblyVersion("1.0.*")]
```

Il carattere `*` viene utilizzato per incrementare automaticamente una porzione della versione automaticamente ogni volta che si compila (spesso utilizzato per il numero "build")

Lettura degli attributi dell'assieme

Utilizzando le API di riflessioni avanzate di .NET, puoi accedere ai metadati di un assembly. Ad esempio, è possibile ottenere l'attributo del titolo di `this` assembly con il seguente codice

```
using System.Linq;
using System.Reflection;

...

Assembly assembly = typeof(this).Assembly;
var titleAttribute = assembly.GetCustomAttributes<AssemblyTitleAttribute>().FirstOrDefault();

Console.WriteLine($"This assembly title is {titleAttribute?.Title}");
```

Controllo delle versioni automatizzato

Il codice nel controllo sorgente ha i numeri di versione per impostazione predefinita (ID SVN o hash Git SHA1) o esplicitamente (tag Git). Anziché aggiornare manualmente le versioni in

AssemblyInfo.cs, è possibile utilizzare un processo di compilazione per scrivere la versione dal sistema di controllo del codice sorgente nei file AssemblyInfo.cs e quindi negli assembly.

I pacchetti GitVersionTask o SemVer.Git.Fody NuGet sono esempi di quanto sopra. Per utilizzare GitVersionTask, ad esempio, dopo aver installato il pacchetto nel progetto rimuovere gli attributi di `Assembly*Version` dai file AssemblyInfo.cs. Questo mette GitVersionTask a capo della versione degli assembly.

Si noti che il Semantic Versioning è sempre più lo standard di *fatto* , quindi questi metodi raccomandano l'uso di tag di controllo sorgente che seguono SemVer.

Campi comuni

È buona norma completare i campi predefiniti di AssemblyInfo. Le informazioni possono essere recuperate dagli installatori e verranno visualizzate quando si utilizza Programmi e funzionalità (Windows 10) per disinstallare o modificare un programma.

Il minimo dovrebbe essere:

- AssemblyTitle - solitamente lo spazio dei nomi, *ovvero* MyCompany.MySolution.MyProject
- AssemblyCompany: il nome completo delle entità legali
- AssemblyProduct - marketing può avere una vista qui
- AssemblyCopyright: tienilo aggiornato perché altrimenti potrebbe sembrare trasandato

'AssemblyTitle' diventa la 'Descrizione file' quando si esamina la scheda Dettagli proprietà della DLL.

[AssemblyConfiguration]

AssemblyConfiguration: l'attributo AssemblyConfiguration deve avere la configurazione utilizzata per creare l'assembly. Utilizzare la compilazione condizionale per includere correttamente diverse configurazioni di assieme. Usa il blocco simile all'esempio qui sotto. Aggiungi tutte le diverse configurazioni che usi di solito.

```
#if (DEBUG)

[assembly: AssemblyConfiguration("Debug")]

#else

[assembly: AssemblyConfiguration("Release")]

#endif
```

[InternalsVisibleTo]

Se si desidera rendere `internal` classi `internal` o le funzioni di un assembly accessibili da un altro assembly, dichiararlo da `InternalsVisibleTo` e il nome dell'assembly a cui è consentito l'accesso.

In questo esempio di codice nell'assembly `MyAssembly.UnitTests` è consentito chiamare elementi

`internal` da `MyAssembly` .

```
[assembly: InternalsVisibleTo("MyAssembly.UnitTests")]
```

Ciò è particolarmente utile per i test unitari per evitare dichiarazioni `public` non necessarie.

[AssemblyKeyFile]

Ogni volta che vogliamo installare il nostro assembly in GAC, è necessario avere un nome sicuro. Per un assembly di denominazione forte, dobbiamo creare una chiave pubblica. Per generare il file `.snk` .

Per creare un file chiave con nome sicuro

1. Prompt dei comandi per sviluppatori per VS2015 (con accesso amministratore)
2. Al prompt dei comandi, digitare cd C: \ Directory_Name e premere INVIO.
3. Al prompt dei comandi, digitare sn -k KeyFileName.snk e quindi premere INVIO.

una volta che il file keyFileName.snk viene creato nella directory specificata, fornire refernce nel progetto. Assegna a `AssemblyKeyFileAttribute` il percorso del file `snk` per generare la chiave quando costruiamo la nostra libreria di classi.

Proprietà -> AssemblyInfo.cs

```
[assembly: AssemblyKeyFile(@"c:\Directory_Name\KeyFileName.snk")]
```

Thi creerà un assembly con un nome forte dopo la build. Dopo aver creato il tuo assembly nome sicuro, puoi installarlo in GAC

Happy Coding :)

Leggi AssemblyInfo.cs Esempi online: https://riptutorial.com/it/csharp/topic/4264/assemblyinfo-cs-esempi

Capitolo 12: Async-Await

introduzione

In C #, un metodo dichiarato `async` non bloccherà all'interno di un processo sincrono, nel caso in cui si stiano utilizzando operazioni basate sull'I / O (ad esempio accesso Web, utilizzo di file, ...). Il risultato di tali metodi asincroni contrassegnati può essere atteso tramite l'uso della parola chiave `await` .

Osservazioni

Un metodo `async` può restituire `void` , `Task` o `Task<T>` .

Il tipo di ritorno `Task` attenderà che il metodo finisca e il risultato sarà `void` . `Task<T>` restituirà un valore da tipo `T` dopo il completamento del metodo.

`async` metodi `async` dovrebbero restituire `Task` o `Task<T>` , in contrasto con il `void` , in quasi tutte le circostanze. non è possibile `await` metodi `async void` , il che porta a una serie di problemi. L'unico scenario in cui un `async` deve restituire `void` è nel caso di un gestore di eventi.

`async` / `await` funziona trasformando il tuo metodo `async` in una macchina a stati. Lo fa creando una struttura dietro le quinte che memorizza lo stato corrente e qualsiasi contesto (come le variabili locali) e espone un metodo `MoveNext()` per far avanzare gli stati (ed eseguire qualsiasi codice associato) ogni volta che un attendibile atteso si completa.

Examples

Semplici chiamate consecutive

```
public async Task<JobResult> GetDataFromWebAsync()
{
  var nextJob = await _database.GetNextJobAsync();
  var response = await _httpClient.GetAsync(nextJob.Uri);
  var pageContents = await response.Content.ReadAsStringAsync();
  return await _database.SaveJobResultAsync(pageContents);
}
```

La cosa principale da notare qui è che mentre ogni `await` metodo -ed è chiamato in modo asincrono - e per il tempo di quella chiamata il controllo viene ceduto al sistema - il flusso all'interno del metodo è lineare e non richiede alcun trattamento speciale a causa di asincronia. Se uno dei metodi chiamati fallisce, l'eccezione verrà elaborata "come previsto", che in questo caso significa che l'esecuzione del metodo verrà interrotta e l'eccezione salirà nello stack.

Try / catch / finally

6.0

A partire da C # 6.0, la parola chiave `await` ora può essere utilizzata all'interno di un blocco `catch` e `finally` .

```
try {
   var client = new AsyncClient();
   await client.DoSomething();
} catch (MyException ex) {
   await client.LogExceptionAsync();
   throw;
} finally {
   await client.CloseAsync();
}
```

5.0 6.0

Prima di C # 6.0, avresti bisogno di fare qualcosa sulla falsariga di quanto segue. Si noti che 6.0 ha anche ripulito i controlli null con l' operatore Null Propagating .

```
AsynClient client;
MyException caughtException;
try {
     client = new AsyncClient();
     await client.DoSomething();
} catch (MyException ex) {
     caughtException = ex;
}

if (client != null) {
    if (caughtException != null) {
       await client.LogExceptionAsync();
    }
    await client.CloseAsync();
    if (caughtException != null) throw caughtException;
}
```

Si noti che se si attende un'attività non creata da `async` (ad esempio un'attività creata da `Task.Run`), alcuni debugger potrebbero interrompersi sulle eccezioni generate dall'attività anche se apparentemente gestite dal try / catch circostante. Questo accade perché il debugger lo considera non gestito rispetto al codice utente. In Visual Studio, c'è un'opzione chiamata "Just My Code" , che può essere disattivata per impedire che il debugger si rompa in tali situazioni.

Installazione di Web.config su target 4.5 per il corretto comportamento asincrono.

Il web.config system.web.httpRuntime deve targetizzare 4.5 per garantire che il thread affitterà il contesto della richiesta prima di riprendere il metodo asincrono.

```
<httpRuntime targetFramework="4.5" />
```

Async e attendere hanno un comportamento indefinito su ASP.NET precedente alla 4.5. Async / await riprenderà su un thread arbitrario che potrebbe non avere il contesto della richiesta. Le applicazioni sotto carico falliranno casualmente con eccezioni di riferimento null accedendo a

HttpContext dopo l'attesa. L'utilizzo di HttpContext.Current in WebApi è pericoloso a causa di async

Chiamate contemporanee

È possibile attendere più chiamate contemporaneamente, invocando dapprima le attività attendibili e *quindi* aspettandole.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await firstTask;
    await secondTask;
}
```

In alternativa, `Task.WhenAll` può essere utilizzato per raggruppare più attività in una singola `Task` , che viene completata quando tutte le attività passate sono complete.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await Task.WhenAll(firstTask, secondTask);
}
```

Puoi anche farlo all'interno di un ciclo, ad esempio:

```
List<Task> tasks = new List<Task>();
while (something) {
    // do stuff
    Task someAsyncTask = someAsyncMethod();
    tasks.Add(someAsyncTask);
}

await Task.WhenAll(tasks);
```

Per ottenere risultati da un'attività dopo aver atteso più attività con Task.WhenAll, è sufficiente attendere di nuovo l'attività. Poiché l'attività è già completata, verrà restituito il risultato

```
var task1 = SomeOpAsync();
var task2 = SomeOtherOpAsync();

await Task.WhenAll(task1, task2);

var result = await task2;
```

Inoltre, `Task.WhenAny` può essere utilizzato per eseguire più attività in parallelo, come `Task.WhenAll` sopra, con la differenza che questo metodo verrà completato quando verrà completata *una* delle attività fornite.

```
public async Task RunConcurrentTasksWhenAny()
{
    var firstTask = TaskOperation("#firstTask executed");
    var secondTask = TaskOperation("#secondTask executed");
    var thirdTask = TaskOperation("#thirdTask executed");
    await Task.WhenAny(firstTask, secondTask, thirdTask);
}
```

L' `Task` restituita da `RunConcurrentTasksWhenAny` verrà completata quando una delle operazioni `firstTask` , `secondTask` o `thirdTask` completata.

Attendere l'operatore e la parola chiave asincrona

`await` operatore e la parola chiave `async` si uniscano:

> Il metodo asincrono in cui viene utilizzata l' **attesa** deve essere modificato dalla parola chiave **async** .

L'opposto non è sempre vero: puoi contrassegnare un metodo come `async` senza utilizzare l' `await` nel suo corpo.

Ciò che in realtà `await` è sospendere l'esecuzione del codice fino al completamento dell'attività atteso; qualsiasi attività può essere attesa.

Nota: non è possibile attendere il metodo asincrono che non restituisce nulla (nulla).

In realtà, la parola 'suspends' è un po 'fuorviante perché non solo l'esecuzione si arresta, ma il thread può diventare libero per l'esecuzione di altre operazioni. Sotto il cofano, l' `await` è implementata da un po 'di magia del compilatore: divide un metodo in due parti - prima e dopo l' `await` . L'ultima parte viene eseguita quando l'attività attesa viene completata.

Se ignoriamo alcuni dettagli importanti, il compilatore lo fa grosso modo per te:

```
public async Task<TResult> DoIt()
{
    // do something and acquire someTask of type Task<TSomeResult>
    var awaitedResult = await someTask;
    // ... do something more and produce result of type TResult
    return result;
}
```

diventa:

```
public Task<TResult> DoIt()
{
    // ...
    return someTask.ContinueWith(task => {
        var result = ((Task<TSomeResult>)task).Result;
        return DoIt_Continuation(result);
    });
}

private TResult DoIt_Continuation(TSomeResult awaitedResult)
```

```
{
    // ...
}
```

Qualsiasi metodo usuale può essere trasformato in asincrono nel seguente modo:

```
await Task.Run(() => YourSyncMethod());
```

Ciò può essere vantaggioso quando è necessario eseguire un metodo a esecuzione prolungata sul thread dell'interfaccia utente senza bloccare l'interfaccia utente.

Ma c'è un'osservazione molto importante qui: **Asincrono non significa sempre concomitante (parallelo o addirittura multi-thread).** Anche su un singolo thread, `async` - `await` consente ancora il codice asincrono. Ad esempio, vedere questo Utilità di pianificazione personalizzata. Un programmatore di compiti così "pazzo" può semplicemente trasformare compiti in funzioni chiamate all'interno dell'elaborazione del loop di messaggi.

Dobbiamo chiederci: quale thread eseguirà la continuazione del nostro metodo `DoIt_Continuation` ?

Per impostazione predefinita, l'operatore di `await` pianifica l'esecuzione della continuazione con il contesto di sincronizzazione corrente. Significa che per impostazione predefinita per le sequenze di esecuzione WinForms e WPF nel thread dell'interfaccia utente. Se, per qualche motivo, è necessario modificare questo comportamento, utilizzare il metodo `Task.ConfigureAwait()` :

```
await Task.Run(() => YourSyncMethod()).ConfigureAwait(continueOnCapturedContext: false);
```

Restituzione di un'attività senza attendere

I metodi che eseguono operazioni asincrone non devono essere utilizzati `await` se:

- C'è solo una chiamata asincrona all'interno del metodo
- La chiamata asincrona è alla fine del metodo
- L'eccezione di cattura / gestione che può verificarsi all'interno dell'attività non è necessaria

Considerate questo metodo che restituisce un `Task` :

```
public async Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;

    return await dataStore.GetByKeyAsync(lookupKey);
}
```

Se `GetByKeyAsync` ha la stessa firma di `GetUserAsync` (restituendo un'attività `Task<User>`), il metodo può essere semplificato:

```
public Task<User> GetUserAsync(int id)
{
```

```
    var lookupKey = "Users" + id;

    return dataStore.GetByKeyAsync(lookupKey);
}
```

In questo caso, il metodo non deve essere contrassegnato come `async` , anche se sta preformando un'operazione asincrona. L'attività restituita da `GetByKeyAsync` viene passata direttamente al metodo chiamante, dove sarà `await` ed.

Importante : se si restituisce l' `Task` invece di attenderla, cambia il comportamento dell'eccezione del metodo, poiché non getterà l'eccezione all'interno del metodo che avvia l'attività ma nel metodo che la attende.

```
public Task SaveAsync()
{
    try {
        return dataStore.SaveChangesAsync();
    }
    catch(Exception ex)
    {
        // this will never be called
        logger.LogException(ex);
    }
}

// Some other code calling SaveAsync()

// If exception happens, it will be thrown here, not inside SaveAsync()
await SaveAsync();
```

Ciò migliorerà le prestazioni poiché salverà il compilatore la generazione di una macchina di stato **asincrono** extra.

Il blocco del codice asincrono può causare deadlock

È una cattiva pratica bloccare le chiamate asincrone poiché può causare deadlock in ambienti con un contesto di sincronizzazione. La migliore pratica è usare async / attendere "fino in fondo". Ad esempio, il seguente codice Windows Form causa un deadlock:

```
private async Task<bool> TryThis()
{
    Trace.TraceInformation("Starting TryThis");
    await Task.Run(() =>
    {
        Trace.TraceInformation("In TryThis task");
        for (int i = 0; i < 100; i++)
        {
            // This runs successfully - the loop runs to completion
            Trace.TraceInformation("For loop " + i);
            System.Threading.Thread.Sleep(10);
        }
    });

    // This never happens due to the deadlock
    Trace.TraceInformation("About to return");
```

```
    return true;
}

// Button click event handler
private void button1_Click(object sender, EventArgs e)
{
    // .Result causes this to block on the asynchronous call
    bool result = TryThis().Result;
    // Never actually gets here
    Trace.TraceInformation("Done with result");
}
```

In sostanza, una volta completata la chiamata asincrona, attende che il contesto di sincronizzazione sia disponibile. Tuttavia, il gestore eventi "trattiene" il contesto di sincronizzazione mentre è in attesa del `TryThis()` metodo `TryThis()` , causando quindi un'attesa circolare.

Per risolvere questo problema, il codice dovrebbe essere modificato in

```
private async void button1_Click(object sender, EventArgs e)
{
  bool result = await TryThis();
  Trace.TraceInformation("Done with result");
}
```

Nota: i gestori di eventi sono l'unico posto in cui dovrebbe essere usato il `async void` (perché non è possibile attendere un metodo di `async void`).

Async / await migliorerà le prestazioni solo se consente alla macchina di eseguire ulteriori operazioni

Considera il seguente codice:

```
public async Task MethodA()
{
     await MethodB();
     // Do other work
}

public async Task MethodB()
{
     await MethodC();
     // Do other work
}

public async Task MethodC()
{
     // Or await some other async work
     await Task.Delay(100);
}
```

Questo non funzionerà meglio di

```
public void MethodA()
```

```
{
     MethodB();
     // Do other work
}

public void MethodB()
{
     MethodC();
     // Do other work
}

public void MethodC()
{
     Thread.Sleep(100);
}
```

Lo scopo principale di async / await è di consentire alla macchina di eseguire operazioni aggiuntive, ad esempio per consentire al thread chiamante di eseguire altri lavori mentre è in attesa di un risultato da qualche operazione di I / O. In questo caso, il thread chiamante non ha mai il permesso di fare più lavoro di quanto sarebbe stato in grado di fare altrimenti, quindi non ci sono guadagni di prestazioni semplicemente chiamando `MethodA()` , `MethodB()` e `MethodC()` sincrono.

Leggi Async-Await online: https://riptutorial.com/it/csharp/topic/48/async-await

Capitolo 13: attributi

Examples

Creare un attributo personalizzato

```
//1) All attributes should be inherited from System.Attribute
//2) You can customize your attribute usage (e.g. place restrictions) by using
System.AttributeUsage Attribute
//3) You can use this attribute only via reflection in the way it is supposed to be used
//4) MethodMetadataAttribute is just a name. You can use it without "Attribute" postfix - e.g.
[MethodMetadata("This text could be retrieved via reflection")].
//5) You can overload an attribute constructors
[System.AttributeUsage(System.AttributeTargets.Method | System.AttributeTargets.Class)]
public class MethodMetadataAttribute : System.Attribute
{
    //this is custom field given just for an example
    //you can create attribute without any fields
    //even an empty attribute can be used - as marker
    public string Text { get; set; }

    //this constructor could be used as [MethodMetadata]
    public MethodMetadataAttribute ()
    {
    }

    //This constructor could be used as [MethodMetadata("String")]
    public MethodMetadataAttribute (string text)
    {
        Text = text;
    }
}
```

Utilizzando un attributo

```
[StackDemo(Text = "Hello, World!")]
public class MyClass
{
    [StackDemo("Hello, World!")]
    static void MyMethod()
    {
    }
}
```

Leggere un attributo

Metodo `GetCustomAttributes` restituisce una serie di attributi personalizzati applicati al membro. Dopo aver recuperato questo array è possibile cercare uno o più attributi specifici.

```
var attribute = typeof(MyClass).GetCustomAttributes().OfType<MyCustomAttribute>().Single();
```

Oppure scorrere attraverso di loro

```
foreach(var attribute in typeof(MyClass).GetCustomAttributes()) {
    Console.WriteLine(attribute.GetType());
}
```

`GetCustomAttribute` metodo di estensione `GetCustomAttribute` di `System.Reflection.CustomAttributeExtensions` recupera un attributo personalizzato di un tipo specificato, può essere applicato a qualsiasi `MemberInfo` .

```
var attribute = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute));
```

`GetCustomAttribute` ha anche una firma generica per specificare il tipo di attributo da cercare.

```
var attribute = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>();
```

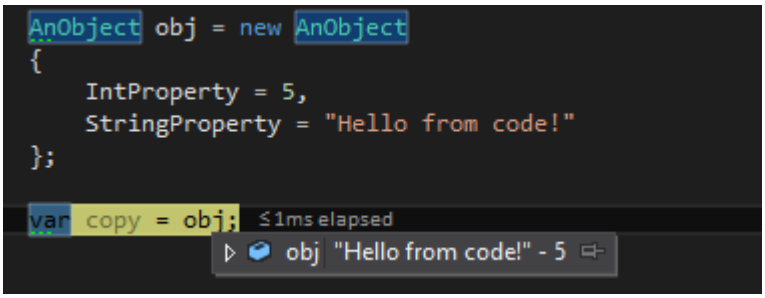
L'argomento Boolean `inherit` può essere passato ad entrambi questi metodi. Se questo valore è impostato su `true` gli antenati dell'elemento dovrebbero essere esaminati.

DebuggerDisplay Attribute

Aggiungere l'attributo `DebuggerDisplay` cambierà il modo in cui il debugger visualizza la classe quando viene posizionata al passaggio del mouse.

Le espressioni racchiuse in `{}` verranno valutate dal debugger. Può trattarsi di una proprietà semplice come nell'esempio seguente o in una logica più complessa.

```
[DebuggerDisplay("{StringProperty} - {IntProperty}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }
}
```



Aggiungendo `,nq` prima della parentesi di chiusura rimuove le virgolette quando si stampa una stringa.

```
[DebuggerDisplay("{StringProperty,nq} - {IntProperty}")]
```

Anche se le espressioni generali sono consentite in `{}` non sono raccomandate. L'attributo `DebuggerDisplay` verrà scritto nei metadati dell'assieme sotto forma di stringa. Le espressioni in `{}`

non vengono verificate per la validità. Quindi un attributo `DebuggerDisplay` contenente una logica più complessa di una semplice aritmetica potrebbe funzionare bene in C #, ma la stessa espressione valutata in VB.NET probabilmente non sarà sintatticamente valida e produrrà un errore durante il debug.

Un modo per rendere `DebuggerDisplay` più indipendente dalla lingua è scrivere l'espressione in un metodo o in una proprietà e chiamarla invece.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }

   private string DebuggerDisplay()
    {
        return $"{StringProperty} - {IntProperty}"";
    }
}
```

Si potrebbe desiderare che `DebuggerDisplay` restituisca tutte o solo alcune proprietà e durante il debug e l'ispezione del tipo dell'oggetto.
L'esempio seguente circonda anche il metodo helper con `#if DEBUG` dato che `DebuggerDisplay` viene utilizzato negli ambienti di debug.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }

#if DEBUG
   private string DebuggerDisplay()
    {
        return
            $"ObjectId:{this.ObjectId}, StringProperty:{this.StringProperty},
Type:{this.GetType()}";
    }
    #endif
}
```

Attributi di informazioni sul chiamante

Gli attributi di informazioni sul chiamante possono essere utilizzati per trasmettere informazioni sull'invocatore al metodo richiamato. La dichiarazione assomiglia a questo:

```
using System.Runtime.CompilerServices;

public void LogException(Exception ex,
                         [CallerMemberName]string callerMemberName = "",
                         [CallerLineNumber]int callerLineNumber = 0,
                         [CallerFilePath]string callerFilePath = "")
```

```
{
    //perform logging
}
```

E l'invocazione assomiglia a questo:

```
public void Save(DBContext context)
{
    try
    {
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        LogException(ex);
    }
}
```

Si noti che solo il primo parametro viene passato esplicitamente al metodo `LogException` mentre il resto di essi verrà fornito in fase di compilazione con i valori pertinenti.

Il parametro `callerMemberName` riceverà il valore `"Save"` , il nome del metodo di chiamata.

Il parametro `callerLineNumber` riceverà il numero di qualsiasi riga su `LogException` è scritta la chiamata del metodo `LogException` .

E il parametro 'callerFilePath' riceverà il percorso completo del file in cui è stato dichiarato il metodo `Save` .

Lettura di un attributo dall'interfaccia

Non esiste un modo semplice per ottenere attributi da un'interfaccia, poiché le classi non ereditano gli attributi da un'interfaccia. Ogni volta che si implementa un'interfaccia o si sovrascrivono i membri in una classe derivata, è necessario dichiarare nuovamente gli attributi. Quindi nell'esempio seguente l'output sarebbe `True` in tutti e tre i casi.

```
using System;
using System.Linq;
using System.Reflection;

namespace InterfaceAttributesDemo {

    [AttributeUsage(AttributeTargets.Interface, Inherited = true)]
    class MyCustomAttribute : Attribute {
        public string Text { get; set; }
    }

    [MyCustomAttribute(Text = "Hello from interface attribute")]
    interface IMyClass {
        void MyMethod();
    }

    class MyClass : IMyClass {
        public void MyMethod() { }
```

```
    }

    public class Program {
        public static void Main(string[] args) {
            GetInterfaceAttributeDemo();
        }

        private static void GetInterfaceAttributeDemo() {
            var attribute1 = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute), true);
            Console.WriteLine(attribute1 == null); // True

            var attribute2 =
typeof(MyClass).GetCustomAttributes(true).OfType<MyCustomAttribute>().SingleOrDefault();
            Console.WriteLine(attribute2 == null); // True

            var attribute3 = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>(true);
            Console.WriteLine(attribute3 == null); // True
        }
    }
}
```

Un modo per recuperare gli attributi dell'interfaccia è cercarli attraverso tutte le interfacce implementate da una classe.

```
var attribute = typeof(MyClass).GetInterfaces().SelectMany(x =>
x.GetCustomAttributes().OfType<MyCustomAttribute>()).SingleOrDefault();
Console.WriteLine(attribute == null); // False
Console.WriteLine(attribute.Text); // Hello from interface attribute
```

Attributo obsoleto

System.Obsolete è un attributo che viene utilizzato per contrassegnare un tipo o un membro che ha una versione migliore e pertanto non deve essere utilizzato.

```
[Obsolete("This class is obsolete. Use SomeOtherClass instead.")]
class SomeClass
{
    //
}
```

Nel caso in cui la classe sopra sia usata, il compilatore darà l'avvertimento "Questa classe è obsoleta. Usa SomeOtherClass invece."

Leggi attributi online: https://riptutorial.com/it/csharp/topic/1062/attributi

Capitolo 14: BackgroundWorker

Sintassi

- `bgWorker.CancellationPending //returns whether the bgWorker was cancelled during its operation`
- `bgWorker.IsBusy //returns true if the bgWorker is in the middle of an operation`
- `bgWorker.ReportProgress(int x) //Reports a change in progress. Raises the "ProgressChanged" event`
- `bgWorker.RunWorkerAsync() //Starts the BackgroundWorker by raising the "DoWork" event`
- `bgWorker.CancelAsync() //instructs the BackgroundWorker to stop after the completion of a task.`

Osservazioni

L'esecuzione di operazioni a esecuzione prolungata all'interno del thread dell'interfaccia utente può causare la mancata risposta dell'applicazione, visualizzando all'utente che ha smesso di funzionare. È preferibile che queste attività vengano eseguite su un thread in background. Una volta completata, l'interfaccia utente può essere aggiornata.

Apportare modifiche all'interfaccia utente durante l'operazione di BackgroundWorker richiede il richiamo delle modifiche al thread dell'interfaccia utente, in genere utilizzando il metodo Control.Invoke sul controllo che si sta aggiornando. Trascurare di farlo farà sì che il tuo programma generi un'eccezione.

In genere, BackgroundWorker viene utilizzato solo nelle applicazioni Windows Form. Nelle applicazioni WPF, le attività vengono utilizzate per scaricare il lavoro sui thread in background (eventualmente in combinazione con async / await). Gli aggiornamenti di marshalling sul thread dell'interfaccia utente vengono in genere eseguiti automaticamente, quando la proprietà da aggiornare implementa INotifyPropertyChanged o manualmente utilizzando il Dispatcher del thread dell'interfaccia utente.

Examples

Assegnazione di gestori di eventi a un BackgroundWorker

Una volta che l'istanza di BackgroundWorker è stata dichiarata, è necessario assegnare le proprietà e i gestori di eventi per le attività che esegue.

```
    /* This is the backgroundworker's "DoWork" event handler. This
       method is what will contain all the work you
       wish to have your program perform without blocking the UI. */

    bgWorker.DoWork += bgWorker_DoWork;
```

```
    /*This is how the DoWork event method signature looks like:*/
    private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
    {
        // Work to be done here
        // ...
        // To get a reference to the current Backgroundworker:
        BackgroundWorker worker = sender as BackgroundWorker;
        // The reference to the BackgroundWorker is often used to report progress
        worker.ReportProgress(...);
    }

    /*This is the method that will be run once the BackgroundWorker has completed its tasks */

    bgWorker.RunWorkerCompleted += bgWorker_CompletedWork;

    /*This is how the RunWorkerCompletedEvent event method signature looks like:*/
    private void bgWorker_CompletedWork(object sender, RunWorkerCompletedEventArgs e)
    {
        // Things to be done after the backgroundworker has finished
    }

   /* When you wish to have something occur when a change in progress
     occurs, (like the completion of a specific task) the "ProgressChanged"
     event handler is used. Note that ProgressChanged events may be invoked
     by calls to bgWorker.ReportProgress(...) only if bgWorker.WorkerReportsProgress
     is set to true.  */

     bgWorker.ProgressChanged += bgWorker_ProgressChanged;

    /*This is how the ProgressChanged event method signature looks like:*/
    private void bgWorker_ProgressChanged(object sender, ProgressChangedEventArgs e)
    {
        // Things to be done when a progress change has been reported

        /* The ProgressChangedEventArgs gives access to a percentage,
         allowing for easy reporting of how far along a process is*/
        int progress = e.ProgressPercentage;
    }
```

Assegnazione di proprietà a un BackgroundWorker

Ciò consente a BackgroundWorker di essere annullato tra le attività

```
bgWorker.WorkerSupportsCancellation = true;
```

Ciò consente al lavoratore di segnalare lo stato di avanzamento tra il completamento delle attività ...

```
bgWorker.WorkerReportsProgress = true;

//this must also be used in conjunction with the ProgressChanged event
```

Creazione di una nuova istanza di BackgroundWorker

Un BackgroundWorker viene comunemente utilizzato per eseguire attività, a volte in termini di

tempo, senza bloccare il thread dell'interfaccia utente.

```
// BackgroundWorker is part of the ComponentModel namespace.
using System.ComponentModel;

namespace BGWorkerExample
{
     public partial class ExampleForm : Form
     {

      // the following creates an instance of the BackgroundWorker named "bgWorker"
      BackgroundWorker bgWorker = new BackgroundWorker();

      public ExampleForm() { ...
```

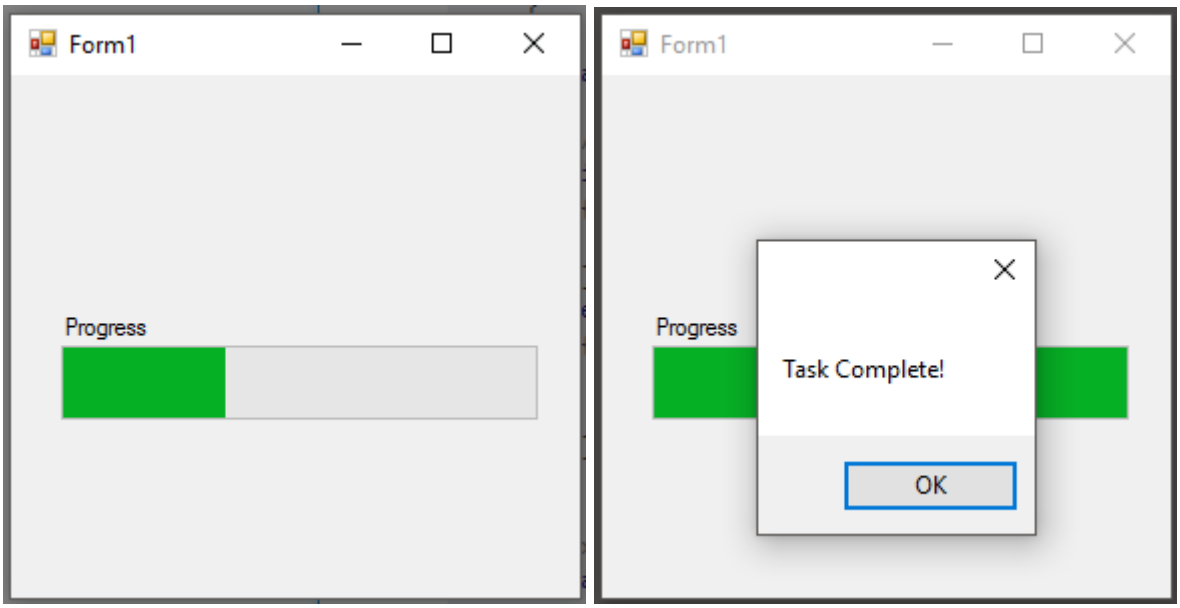
Utilizzo di BackgroundWorker per completare un'attività.

Nell'esempio seguente viene illustrato l'utilizzo di BackgroundWorker per l'aggiornamento di una barra di avanzamento di WinForms. Lo sfondoWorker aggiornerà il valore della barra di avanzamento senza bloccare il thread dell'interfaccia utente, mostrando così un'interfaccia utente reattiva mentre il lavoro viene eseguito in background.

```
namespace BgWorkerExample
{
    public partial class Form1 : Form
{

    //a new instance of a backgroundWorker is created.
    BackgroundWorker bgWorker = new BackgroundWorker();

    public Form1()
    {
        InitializeComponent();

        prgProgressBar.Step = 1;

        //this assigns event handlers for the backgroundWorker
        bgWorker.DoWork += bgWorker_DoWork;
        bgWorker.RunWorkerCompleted += bgWorker_WorkComplete;

        //tell the backgroundWorker to raise the "DoWork" event, thus starting it.
        //Check to make sure the background worker is not already running.
        if(!bgWorker.IsBusy)
            bgWorker.RunWorkerAsync();

    }

    private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
    {
        //this is the method that the backgroundworker will perform on in the background
thread.
        /* One thing to note! A try catch is not necessary as any exceptions will terminate
the backgroundWorker and report
          the error to the "RunWorkerCompleted" event */
        CountToY();
    }
```

```
    private void bgWorker_WorkComplete(object sender, RunWorkerCompletedEventArgs e)
    {
        //e.Error will contain any exceptions caught by the backgroundWorker
        if (e.Error != null)
        {
            MessageBox.Show(e.Error.Message);
        }
        else
        {
            MessageBox.Show("Task Complete!");
            prgProgressBar.Value = 0;
        }
    }

    // example method to perform a "long" running task.
    private void CountToY()
    {
        int x = 0;

        int maxProgress = 100;
        prgProgressBar.Maximum = maxProgress;


        while (x < maxProgress)
        {
            System.Threading.Thread.Sleep(50);
            Invoke(new Action(() => { prgProgressBar.PerformStep(); }));
            x += 1;
        }
    }


}
```

Il risultato è il seguente ...



Leggi BackgroundWorker online: https://riptutorial.com/it/csharp/topic/1588/backgroundworker

Capitolo 15: BigInteger

Osservazioni

Quando usare

`BigInteger` oggetti `BigInteger` sono per loro natura molto pesanti sulla RAM. Di conseguenza, dovrebbero essere usati solo quando assolutamente necessario, cioè per numeri su scala veramente astronomica.

Oltre a ciò, tutte le operazioni aritmetiche su questi oggetti sono di un ordine di grandezza più lento delle loro controparti primitive, questo problema viene ulteriormente aggravato man mano che il numero cresce man mano che non sono di dimensioni fisse. È quindi possibile che un `BigInteger` possa causare un arresto anomalo consumando tutta la RAM disponibile.

alternative

Se la velocità è imperativa per la soluzione, potrebbe essere più efficiente implementare questa funzionalità da soli utilizzando una classe che include un `Byte[]` e sovraccaricando gli operatori necessari da soli. Tuttavia, questo richiede un notevole sforzo extra.

Examples

Calcola il primo numero di Fibonacci a 1.000 cifre

Includere `using System.Numerics` e aggiungere un riferimento a `System.Numerics` al progetto.

```
using System;
using System.Numerics;

namespace Euler_25
{
    class Program
    {
        static void Main(string[] args)
        {
            BigInteger l1 = 1;
            BigInteger l2 = 1;
            BigInteger current = l1 + l2;
            while (current.ToString().Length < 1000)
            {
                l2 = l1;
                l1 = current;
                current = l1 + l2;
            }
            Console.WriteLine(current);
        }
    }
}
```

Questo semplice algoritmo esegue l'iterazione dei numeri di Fibonacci fino a raggiungere una cifra di almeno 1000 cifre decimali, quindi lo stampa. Questo valore è significativamente più grande di `ulong` potrebbe mantenere un `ulong` .

Teoricamente, l'unico limite della classe `BigInteger` è la quantità di RAM che l'applicazione può consumare.

Nota: `BigInteger` è disponibile solo in .NET 4.0 e versioni successive.

Leggi BigInteger online: https://riptutorial.com/it/csharp/topic/5654/biginteger

Capitolo 16: BindingList

Examples

Evitando l'iterazione N * 2

Questo viene inserito in un gestore di eventi Windows Form

```
var nameList = new BindingList<string>();
ComboBox1.DataSource = nameList;
for(long i = 0; i < 10000; i++ ) {
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });
}
```

Ciò richiede molto tempo per eseguire, per risolvere, fare quanto segue:

```
var nameList = new BindingList<string>();
ComboBox1.DataSource = nameList;
nameList.RaiseListChangedEvents = false;
for(long i = 0; i < 10000; i++ ) {
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });
}
nameList.RaiseListChangedEvents = true;
nameList.ResetBindings();
```

Aggiungi articolo alla lista

```
BindingList<string> listOfUIItems = new BindingList<string>();
listOfUIItems.Add("Alice");
listOfUIItems.Add("Bob");
```

Leggi BindingList online: https://riptutorial.com/it/csharp/topic/182/bindinglist--t-

Capitolo 17: C # 3.0 Caratteristiche

Osservazioni

La versione 3.0 di C # è stata rilasciata come parte di .Net versione 3.5. Molte delle funzionalità aggiunte con questa versione erano a supporto di LINQ (Language Integrated Queries).

Elenco delle funzionalità aggiunte:

- LINQ
- Espressioni Lambda
- Metodi di estensione
- Tipi anonimi
- Variabili implicitamente tipizzate
- Inizializzatori di oggetti e collezioni
- Proprietà implementate automaticamente
- Alberi espressione

Examples

Variabili implicitamente tipizzate (var)

La parola chiave `var` consente a un programmatore di digitare implicitamente una variabile in fase di compilazione. `var` dichiarazioni `var` hanno lo stesso tipo di variabili dichiarate esplicitamente.

```
var squaredNumber = 10 * 10;
var squaredNumberDouble = 10.0 * 10.0;
var builder = new StringBuilder();
var anonymousObject = new
{
    One = SquaredNumber,
    Two = SquaredNumberDouble,
    Three = Builder
}
```

I tipi delle variabili precedenti sono rispettivamente `int` , `double` , `StringBuilder` e un tipo anonimo.

È importante notare che una variabile `var` non viene digitata in modo dinamico. `SquaredNumber = Builder` non è valido poiché stai provando a impostare `int` su un'istanza di `StringBuilder`

Language Integrated Queries (LINQ)

```
//Example 1
int[] array = { 1, 5, 2, 10, 7 };

// Select squares of all odd numbers in the array sorted in descending order
IEnumerable<int> query = from x in array
                         where x % 2 == 1
```

```
                        orderby x descending
                        select x * x;
// Result: 49, 25, 1
```

Esempio tratto dall'articolo di Wikipedia su C # 3.0, sotto-sezione LINQ

L'esempio 1 utilizza la sintassi della query che è stata progettata per sembrare simile alle query SQL.

```
//Example 2
IEnumerable<int> query = array.Where(x => x % 2 == 1)
    .OrderByDescending(x => x)
    .Select(x => x * x);
// Result: 49, 25, 1 using 'array' as defined in previous example
```

Esempio tratto dall'articolo di Wikipedia su C # 3.0, sotto-sezione LINQ

L'esempio 2 utilizza la sintassi del metodo per ottenere lo stesso risultato dell'esempio 1.

È importante notare che, in C #, la sintassi della query LINQ è zucchero sintattico per la sintassi del metodo LINQ. Il compilatore traduce le query in chiamate di metodo in fase di compilazione. Alcune query devono essere espresse nella sintassi del metodo. Da MSDN : "Ad esempio, è necessario utilizzare una chiamata al metodo per esprimere una query che recupera il numero di elementi che corrispondono a una condizione specificata."

Espressioni Lambda

Lambda Expresions è un'estensione di metodi anonimi che consentono parametri implicitamente tipizzati e valori di ritorno. La loro sintassi è meno prolissa di metodi anonimi e segue uno stile di programmazione funzionale.

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

public class Program
{
    public static void Main()
    {
        var numberList = new List<int> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        var sumOfSquares = numberList.Select( number => number * number )
            .Aggregate( (int first, int second) => { return first + second; } );
        Console.WriteLine( sumOfSquares );
    }
}
```

Il codice sopra mostrerà la somma dei quadrati dei numeri da 1 a 10 alla console.

La prima espressione lambda quadra i numeri nella lista. Poiché esiste solo una parentesi sui parametri, può essere omessa. Puoi includere le parentesi se lo desideri:

```
.Select( (number) => number * number);
```

o digitare esplicitamente il parametro, ma sono necessarie le parentesi:

```
.Select( (int number) => number * number);
```

Il corpo lambda è un'espressione e ha un ritorno implicito. Puoi usare un corpo di statement anche se lo desideri. Questo è utile per lambda più complessi.

```
.Select( number => { return number * number; } );
```

Il metodo select restituisce un nuovo IEnumerable con i valori calcolati.

La seconda espressione lambda somma i numeri nell'elenco restituito dal metodo select. Le parentesi sono obbligatorie in quanto vi sono più parametri. I tipi dei parametri sono digitati in modo esplicito ma non è necessario. Il metodo seguente è equivalente.

```
.Aggregate( (first, second) => { return first + second; } );
```

Come è questo:

```
.Aggregate( (int first, int second) => first + second );
```

Tipi anonimi

I tipi anonimi forniscono un modo conveniente per incapsulare un insieme di proprietà di sola lettura in un singolo oggetto senza dover prima definire esplicitamente un tipo. Il nome del tipo è generato dal compilatore e non è disponibile a livello di codice sorgente. Il tipo di ogni proprietà è dedotto dal compilatore.

È possibile creare tipi anonimi utilizzando la `new` parola chiave seguita da una parentesi graffa *(`{`)* . All'interno delle parentesi graffe, è possibile definire proprietà come nel codice sottostante.

```
var v = new { Amount = 108, Message = "Hello" };
```

È anche possibile creare una serie di tipi anonimi. Vedi il codice qui sotto:

```
var a = new[] {
    new {
        Fruit = "Apple",
        Color = "Red"
    },
    new {
        Fruit = "Banana",
        Color = "Yellow"
    }
};
```

O usalo con le query LINQ:

```
var productQuery = from prod in products
```

```
                    select new { prod.Color, prod.Price };
```

Leggi C # 3.0 Caratteristiche online: https://riptutorial.com/it/csharp/topic/3820/c-sharp-3-0-caratteristiche

Capitolo 18: C # 4.0 Caratteristiche

Examples

Parametri opzionali e argomenti con nome

Possiamo omettere l'argomento nella chiamata se quell'argomento è un argomento facoltativo Ogni argomento facoltativo ha il proprio valore predefinito Prenderà il valore predefinito se non forniamo il valore Un valore predefinito di un argomento facoltativo deve essere un

1. Espressione costante
2. Deve essere un tipo di valore come enum o struct.
3. Deve essere un'espressione di default del modulo (valueType)

Deve essere impostato alla fine dell'elenco dei parametri

Parametri del metodo con valori predefiniti:

```
public void ExampleMethod(int required, string optValue = "test", int optNum = 42)
{
    //...
}
```

Come detto da MSDN, un argomento con nome,

Consente di passare l'argomento alla funzione associando il nome del parametro Non è necessario per ricordare la posizione dei parametri che non siamo a conoscenza di sempre. Non c'è bisogno di guardare l'ordine dei parametri nella lista dei parametri della funzione chiamata. Possiamo specificare il parametro per ogni argomento in base al suo nome.

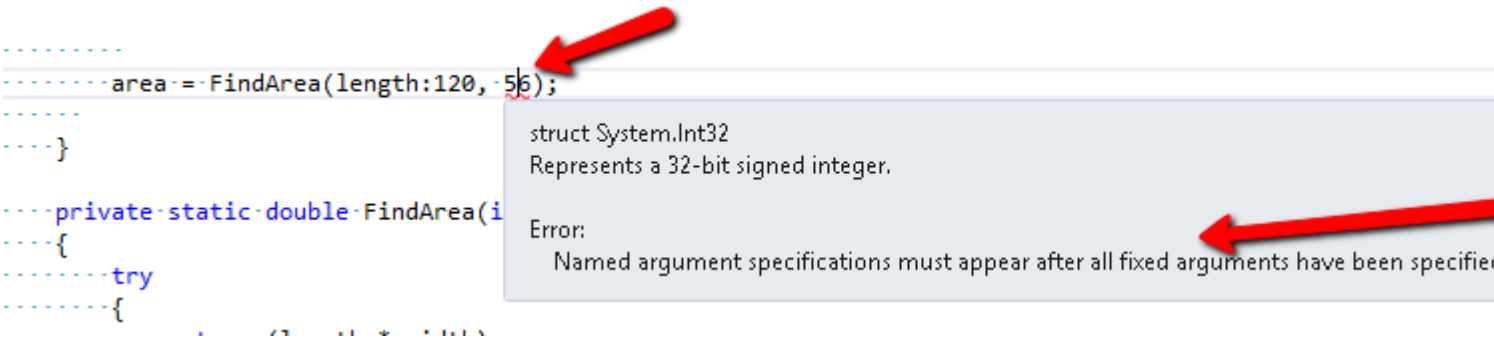
Argomenti con nome:

```
// required = 3, optValue = "test", optNum = 4
ExampleMethod(3, optNum: 4);
// required = 2, optValue = "foo", optNum = 42
ExampleMethod(2, optValue: "foo");
// required = 6, optValue = "bar", optNum = 1
ExampleMethod(optNum: 1, optValue: "bar", required: 6);
```

Limitazione dell'uso di un argomento con nome

La specifica dell'argomento con nome deve apparire dopo che sono stati specificati tutti gli argomenti fissi.

Se si usa un argomento con nome prima di un argomento fisso, si otterrà un errore in fase di compilazione come segue.



La specifica dell'argomento con nome deve apparire dopo che sono stati specificati tutti gli argomenti fissi

Varianza

Interfacce e delegati generici possono avere i loro parametri di tipo contrassegnati come *covarianti* o *controvarianti* usando rispettivamente le parole chiave `out` e `in` . Queste dichiarazioni vengono quindi rispettate per le conversioni di tipo, sia implicite che esplicite, e compilano sia il tempo che il tempo di esecuzione.

Ad esempio, l'interfaccia esistente `IEnumerable<T>` è stata ridefinita come covariante:

```
interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}
```

L'interfaccia esistente IComparer è stata ridefinita come controvariante:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

Parola chiave ref opzionale quando si utilizza COM

La parola chiave ref per i chiamanti di metodi è ora facoltativa quando si chiama nei metodi forniti dalle interfacce COM. Dato un metodo COM con la firma

```
void Increment(ref int x);
```

ora l'invocazione può essere scritta come entrambi

```
Increment(0); // no need for "ref" or a place holder variable any more
```

Ricerca dinamica dei membri

Una nuova `dynamic` pseudo-tipo viene introdotta nel sistema di tipo C #. Viene trattato come `System.Object` , ma in aggiunta, qualsiasi accesso membro (chiamata di metodo, campo, proprietà

o accesso dell'indicizzatore o invocazione di un delegato) o applicazione di un operatore su un valore di tale tipo è consentito senza alcun tipo di controllo, e la sua risoluzione è posticipata fino al momento dell'esecuzione. Questo è noto come digitazione anatra o rilegatura tardiva. Per esempio:

```
// Returns the value of Length property or field of any object
int GetLength(dynamic obj)
{
    return obj.Length;
}

GetLength("Hello, world");        // a string has a Length property,
GetLength(new int[] { 1, 2, 3 }); // and so does an array,
GetLength(42);                    // but not an integer - an exception will be thrown
                                  // in GetLength method at run-time
```

In questo caso, viene utilizzato il tipo dinamico per evitare riflessioni più dettagliate. Usa ancora Reflection sotto il cofano, ma di solito è più veloce grazie al caching.

Questa funzionalità è principalmente rivolta all'interoperabilità con linguaggi dinamici.

```
// Initialize the engine and execute a file
var runtime = ScriptRuntime.CreateFromConfiguration();
dynamic globals = runtime.Globals;
runtime.ExecuteFile("Calc.rb");

// Use Calc type from Ruby
dynamic calc = globals.Calc.@new();
calc.valueA = 1337;
calc.valueB = 666;
dynamic answer = calc.Calculate();
```

Il tipo dinamico ha applicazioni anche nel codice per lo più tipizzato staticamente, per esempio rende possibile la doppia spedizione senza implementare il pattern Visitor.

Leggi C # 4.0 Caratteristiche online: https://riptutorial.com/it/csharp/topic/3093/c-sharp-4-0-caratteristiche

Capitolo 19: C # 5.0 Caratteristiche

Sintassi

- ***Async e attesa***
- public **Task** MyTask **Async** () {doSomething (); }

  attendo MyTaskAsync ();
- public **Task** `<string>` MyStringTask **Async** () {return getSomeString (); }

  string MyString = attendi MyStringTaskAsync ();
- ***Attributi informazioni sul chiamante***
- public void MyCallerAttributes (stringa MyMessage,

  [CallerMemberName] string MemberName = "",

  [CallerFilePath] string SourceFilePath = "",

  [CallerLineNumber] int LineNumber = 0)
- Trace.WriteLine ("My Message:" + MyMessage);

  Trace.WriteLine ("Membro:" + MemberName);

  Trace.WriteLine ("Percorso file di origine:" + SourceFilePath);

  Trace.WriteLine ("Line Number:" + LineNumber);

Parametri

Metodo / modificatore con parametro	Dettagli
`Type<T>`	`T` è il tipo di ritorno

Osservazioni

C # 5.0 è abbinato a Visual Studio .NET 2012

Examples

Async e attesa

`async` e `await` sono due operatori che hanno lo scopo di migliorare le prestazioni liberando Thread e attendendo il completamento delle operazioni prima di andare avanti.

Ecco un esempio di come ottenere una stringa prima di restituire la sua lunghezza:

```
//This method is async because:
//1. It has async and Task or Task<T> as modifiers
//2. It ends in "Async"
async Task<int> GetStringLengthAsync(string URL){
    HttpClient client = new HttpClient();
    //Sends a GET request and returns the response body as a string
    Task<string> getString = client.GetStringAsync(URL);
    //Waits for getString to complete before returning its length
    string contents = await getString;
    return contents.Length;
}

private async void doProcess(){
    int length = await GetStringLengthAsync("http://example.com/");
    //Waits for all the above to finish before printing the number
    Console.WriteLine(length);
}
```

Ecco un altro esempio di download di un file e gestione di ciò che accade quando il progresso è cambiato e al termine del download (ci sono due modi per farlo):

Metodo 1:

```
//This one using async event handlers, but not async coupled with await
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){
    WebClient web = new WebClient();
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    web.DownloadFileCompleted += new AsyncCompletedEventHandler(FileCompleted);
    //Download the file asynchronously
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
}

//event called for when download progress has changed
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){
    //example code
    int i = 0;
    i++;
    doSomething();
}

//event called for when download has finished
private void FileCompleted(object sender, AsyncCompletedEventArgs e){
    Console.WriteLine("Completed!")
}
```

Metodo 2:

```
//however, this one does
//Refer to first example on why this method is async
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){
    WebClient web = new WebClient();
```

```
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    //Download the file async
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
    //Notice how there is no complete event, instead we're using techniques from the first
example
}
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){
    int i = 0;
    i++;
    doSomething();
}
private void doProcess(){
    //Wait for the download to finish
    await DownloadAndUpdateAsync(new Uri("http://example.com/file"))
    doSomething();
}
```

Attributi informazioni sul chiamante

Le CIA sono intese come un modo semplice per ottenere attributi da qualunque cosa stia chiamando il metodo mirato. C'è solo un modo per usarli e ci sono solo 3 attributi.

Esempio:

```
//This is the "calling method": the method that is calling the target method
public void doProcess()
{
    GetMessageCallerAttributes("Show my attributes.");
}
//This is the target method
//There are only 3 caller attributes
public void GetMessageCallerAttributes(string message,
    //gets the name of what is calling this method
    [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",
    //gets the path of the file in which the "calling method" is in
    [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",
    //gets the line number of the "calling method"
    [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)
{
    //Writes lines of all the attributes
    System.Diagnostics.Trace.WriteLine("Message: " + message);
    System.Diagnostics.Trace.WriteLine("Member: " + memberName);
    System.Diagnostics.Trace.WriteLine("Source File Path: " + sourceFilePath);
    System.Diagnostics.Trace.WriteLine("Line Number: " + sourceLineNumber);
}
```

Esempio di output:

```
//Message: Show my attributes.
//Member: doProcess
//Source File Path: c:\Path\To\The\File
//Line Number: 13
```

Leggi C # 5.0 Caratteristiche online: https://riptutorial.com/it/csharp/topic/4584/c-sharp-5-0-caratteristiche

Capitolo 20: C # 6.0 Caratteristiche

introduzione

Questa sesta iterazione del linguaggio C # è fornita dal compilatore di Roslyn. Questo compilatore è uscito con la versione 4.6 di .NET Framework, tuttavia può generare codice in un modo compatibile con le versioni precedenti per consentire il targeting di versioni precedenti del framework. Il codice di versione 6 C # può essere compilato in modo completamente retroattivo con .NET 4.0. Può anche essere utilizzato per i framework precedenti, tuttavia alcune funzionalità che richiedono supporto framework aggiuntivo potrebbero non funzionare correttamente.

Osservazioni

La sesta versione di C # è stata rilasciata a luglio 2015 insieme a Visual Studio 2015 e .NET 4.6.

Oltre ad aggiungere alcune nuove funzionalità linguistiche include una completa riscrittura del compilatore. Precedentemente `csc.exe` era un'applicazione nativa di Win32 scritta in C ++, con C # 6 ora è un'applicazione gestita .NET scritta in C #. Questa riscrittura era conosciuta come progetto "Roslyn" e il codice è ora open source e disponibile su GitHub .

Examples

Nome dell'operatore

L'operatore `nameof` restituisce il nome di un elemento di codice come una `string` . Ciò è utile quando si `INotifyPropertyChanged` eccezioni relative agli argomenti del metodo e anche quando si implementa `INotifyPropertyChanged` .

```
public string SayHello(string greeted)
{
    if (greeted == null)
        throw new ArgumentNullException(nameof(greeted));

    Console.WriteLine("Hello, " + greeted);
}
```

L'operatore `nameof` viene valutato al momento della compilazione e modifica l'espressione in una stringa letterale. Questo è utile anche per le stringhe che prendono il nome dal loro membro che le espone. Considera quanto segue:

```
public static class Strings
{
    public const string Foo = nameof(Foo); // Rather than Foo = "Foo"
    public const string Bar = nameof(Bar); // Rather than Bar = "Bar"
}
```

Dal `nameof` espressioni `nameof` sono costanti in fase di compilazione, possono essere utilizzate in attributi, etichette `case` , istruzioni `switch` e così via.

È conveniente usare `nameof` con `Enum` s. Invece di:

```
Console.WriteLine(Enum.One.ToString());
```

è possibile usare:

```
Console.WriteLine(nameof(Enum.One))
```

L'output sarà `One` in entrambi i casi.

L'operatore `nameof` può accedere ai membri non statici usando la sintassi di tipo statico. Invece di fare:

```
string foo = "Foo";
string lengthName = nameof(foo.Length);
```

Può essere sostituito con:

```
string lengthName = nameof(string.Length);
```

L'output sarà `Length` in entrambi gli esempi. Tuttavia, quest'ultimo impedisce la creazione di istanze non necessarie.

Sebbene l'operatore `nameof` con la maggior parte dei costrutti di linguaggio, esistono alcune limitazioni. Ad esempio, non è possibile utilizzare l'operatore `nameof` tipi generici aperti o valori restituiti dal metodo:

```
public static int Main()
{
    Console.WriteLine(nameof(List<>)); // Compile-time error
    Console.WriteLine(nameof(Main())); // Compile-time error
}
```

Inoltre, se lo si applica a un tipo generico, il parametro di tipo generico verrà ignorato:

```
Console.WriteLine(nameof(List<int>));  // "List"
Console.WriteLine(nameof(List<bool>)); // "List"
```

Per ulteriori esempi, vedere questo argomento dedicato a `nameof` .

Soluzione alternativa per le versioni

precedenti (maggiori dettagli)

Sebbene l'operatore `nameof` non esista in C # per le versioni precedenti alla 6.0, è possibile utilizzare funzionalità simili utilizzando `MemberExpression` come nell'esempio seguente:

6.0

Espressione:

```
public static string NameOf<T>(Expression<Func<T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}

public static string NameOf<TObj, T>(Expression<Func<TObj, T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}
```

Uso:

```
string variableName = NameOf(() => variable);
string propertyName = NameOf((Foo o) => o.Bar);
```

Si noti che questo approccio determina la creazione di un albero di espressioni per ogni chiamata, quindi le prestazioni sono molto peggiori rispetto all'operatore `nameof` che viene valutato in fase di compilazione e che non ha sovraccarico in fase di esecuzione.

Membri della funzione con espressione corporea

I membri delle funzioni con corpo espressivo consentono l'uso di espressioni lambda come corpi membri. Per i membri semplici, può risultare in un codice più pulito e più leggibile.

Le funzioni con corpo espressivo possono essere utilizzate per proprietà, indicizzatori, metodi e operatori.

Proprietà

```
public decimal TotalPrice => BasePrice + Taxes;
```

È equivalente a:

```
public decimal TotalPrice
{
    get
```

```
    {
        return BasePrice + Taxes;
    }
}
```

Quando una funzione con un'espressione viene utilizzata con una proprietà, la proprietà viene implementata come proprietà di solo getter.

Visualizza la demo

---

indicizzatori

```
public object this[string key] => dictionary[key];
```

È equivalente a:

```
public object this[string key]
{
    get
    {
        return dictionary[key];
    }
}
```

---

metodi

```
static int Multiply(int a, int b) => a * b;
```

È equivalente a:

```
static int Multiply(int a, int b)
{
    return a * b;
}
```

Che può essere utilizzato anche con metodi `void` :

```
public void Dispose() => resource?.Dispose();
```

Un override di `ToString` potrebbe essere aggiunto alla classe `Pair<T>` :

```
public override string ToString() => $"{First}, {Second}";
```

Inoltre, questo approccio semplicistico funziona con la parola chiave `override` :

```
public class Foo
{
    public int Bar { get; }

    public string override ToString() => $"Bar: {Bar}";
}
```

---

operatori

Questo può anche essere utilizzato dagli operatori:

```
public class Land
{
    public double Area { get; set; }

    public static Land operator +(Land first, Land second) =>
        new Land { Area = first.Area + second.Area };
}
```

---

limitazioni

I membri delle funzioni con corpo di espressione hanno alcune limitazioni. Non possono contenere istruzioni di blocco e qualsiasi altra istruzione che contenga blocchi: `if` , `switch` , `for` , `foreach` , `while` , `do` , `try` , etc.

Alcuni `if` dichiarazioni possono essere sostituite con operatori ternari. Alcune istruzioni `for` e `foreach` possono essere convertite in query LINQ, ad esempio:

```
IEnumerable<string> Digits
{
    get
    {
        for (int i = 0; i < 10; i++)
            yield return i.ToString();
    }
}
```

```
IEnumerable<string> Digits => Enumerable.Range(0, 10).Select(i => i.ToString());
```

In tutti gli altri casi, è possibile utilizzare la vecchia sintassi per i membri delle funzioni.

I membri delle funzioni con corpo di espressione possono contenere `async` / `await` , ma spesso sono ridondanti:

```
async Task<int> Foo() => await Bar();
```

Può essere sostituito con:

```
Task<int> Foo() => Bar();
```

Filtri di eccezione

I filtri di eccezione offrono agli sviluppatori la possibilità di aggiungere una condizione (sotto forma di espressione `boolean`) a un blocco catch , consentendo l'esecuzione del `catch` solo se la condizione è `true` .

I filtri di eccezione consentono la propagazione delle informazioni di debug nell'eccezione originale, dove come utilizzando un'istruzione `if` all'interno di un blocco `catch` e il richiamo dell'eccezione interrompe la propagazione delle informazioni di debug nell'eccezione originale. Con i filtri delle eccezioni, l'eccezione continua a propagarsi verso l'alto nello stack di chiamate, a *meno che* la condizione non sia soddisfatta. Di conseguenza, i filtri delle eccezioni rendono l'esperienza di debug molto più semplice. Invece di fermarsi sull'istruzione `throw` , il debugger si fermerà sull'istruzione che lancia l'eccezione, mantenendo lo stato corrente e tutte le variabili locali. Le discariche sono coinvolte in modo simile.

> I filtri di eccezione sono stati supportati dal **CLR** sin dall'inizio e sono stati accessibili da VB.NET e F # per oltre un decennio esponendo una parte del modello di gestione delle eccezioni CLR. Solo dopo il rilascio di C # 6.0 la funzionalità è stata disponibile anche per gli sviluppatori C #.

Utilizzo dei filtri delle eccezioni

I filtri di eccezione vengono utilizzati aggiungendo una clausola `when` all'espressione `catch` . È possibile usare qualsiasi espressione che restituisca un `bool` in una clausola `when` (tranne attendere). La variabile Exception dichiarata `ex` è accessibile dall'interno della clausola `when` :

```
var SqlErrorToIgnore = 123;
try
{
    DoSQLOperations();
}
catch (SqlException ex) when (ex.Number != SqlErrorToIgnore)
{
    throw new Exception("An error occurred accessing the database", ex);
}
```

È possibile combinare più blocchi `catch` con clausole `when` . Il primo `when` clausola che restituisce `true` causerà l'intercettazione dell'eccezione. Il suo blocco di `catch` verrà inserito, mentre le altre clausole di `catch` verranno ignorate (le loro clausole `when` non saranno valutate). Per esempio:

```
try
{ ... }
catch (Exception ex) when (someCondition) //If someCondition evaluates to true,
                                          //the rest of the catches are ignored.
{ ... }
catch (NotImplementedException ex) when (someMethod()) //someMethod() will only run if
                                                       //someCondition evaluates to false
```

```
{ ... }
catch(Exception ex) // If both when clauses evaluate to false
{ ... }
```

Risky quando la clausola

> **Attenzione**
>
> Può essere rischioso utilizzare filtri di eccezione: quando l' `Exception` viene lanciata dall'interno del `when` la clausola, l' `Exception` dal `when` clausola viene ignorato e viene trattato come `false` . Questo approccio consente agli sviluppatori di scrivere `when` clausola senza occuparsi di casi non validi.

Il seguente esempio illustra un tale scenario:

```
public static void Main()
{
    int a = 7;
    int b = 0;
    try
    {
        DoSomethingThatMightFail();
    }
    catch (Exception ex) when (a / b == 0)
    {
        // This block is never reached because a / b throws an ignored
        // DivideByZeroException which is treated as false.
    }
    catch (Exception ex)
    {
        // This block is reached since the DivideByZeroException in the
        // previous when clause is ignored.
    }
}

public static void DoSomethingThatMightFail()
{
    // This will always throw an ArgumentNullException.
    Type.GetType(null);
}
```

Visualizza la demo

Si noti che i filtri delle eccezioni evitano i problemi con il numero di riga confusi associati all'uso di `throw` quando il codice fallito si trova all'interno della stessa funzione. Ad esempio in questo caso il numero di riga è riportato come 6 anziché 3:

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) {
6.     throw;
7. }
```

Il numero di riga di eccezione è segnalato come 6 perché l'errore è stato rilevato e ri-generato con l'istruzione `throw` sulla riga 6.

Lo stesso non succede con i filtri delle eccezioni:

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) when (a != 0) {
6.     throw;
7. }
```

In questo esempio `a` è 0 quindi la clausola `catch` viene ignorata ma 3 viene riportato come numero di riga. Questo perché **non si srotolano lo stack** . Più specificamente, l'eccezione *non viene rilevata* sulla linea 5 perché `a` infatti fa uguale `0` e quindi non v'è alcuna possibilità per l'eccezione di essere ri-gettato sulla linea 6 perché la linea 6 non viene eseguito.

Registrazione come effetto collaterale

Le chiamate al metodo nella condizione possono causare effetti collaterali, pertanto i filtri delle eccezioni possono essere utilizzati per eseguire codice su eccezioni senza rilevarli. Un esempio comune che sfrutta questo è un metodo `Log` che restituisce sempre `false` . Ciò consente di tracciare le informazioni del registro durante il debug senza la necessità di ripetere l'eccezione.

> **Tieni presente che** sebbene questo sembra essere un modo comodo di registrazione, può essere rischioso, specialmente se vengono utilizzati assembly di logging di terze parti. Questi potrebbero generare eccezioni durante l'accesso a situazioni non ovvie che potrebbero non essere rilevate facilmente (vedere **Risky `when(...)` clausola** precedente).

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex) when (Log(ex, "An error occurred"))
{
    // This catch block will never be reached
}

// ...

static bool Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
    return false;
}
```

Visualizza la demo

L'approccio comune nelle versioni precedenti di C # era di registrare e rilanciare l'eccezione.

6.0

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex)
{
     Log(ex, "An error occurred");
     throw;
}

// ...

static void Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
}
```

Visualizza la demo

Il blocco `finally`

Il blocco `finally` viene eseguito ogni volta che l'eccezione viene lanciata o meno. Una sottigliezza con le espressioni in `when` filtri delle eccezioni vengono eseguiti più in alto nello stack *prima di* entrare nei blocchi `finally` interni. Ciò può causare risultati e comportamenti imprevisti quando il codice tenta di modificare lo stato globale (come l'utente o la cultura del thread corrente) e reimpostarlo in un blocco `finally` .

Esempio: `finally` block

```
private static bool Flag = false;

static void Main(string[] args)
{
    Console.WriteLine("Start");
    try
    {
        SomeOperation();
    }
    catch (Exception) when (EvaluatesTo())
    {
        Console.WriteLine("Catch");
    }
    finally
    {
        Console.WriteLine("Outer Finally");
    }
}

private static bool EvaluatesTo()
{
    Console.WriteLine($"EvaluatesTo: {Flag}");
    return true;
```

```
}

private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

Uscita prodotta:

Inizio
Valutazioni: True
Infine interiore
Catturare
Finalmente esterno

Visualizza la demo

Nell'esempio precedente, se il metodo `SomeOperation` non desidera "perdere" lo stato globale cambia in clausole `when` del chiamante, dovrebbe anche contenere un blocco `catch` per modificare lo stato. Per esempio:

```
private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    catch
    {
       Flag = false;
       throw;
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

È anche comune vedere classi di helper `IDisposable` sfruttano la semantica dell'utilizzo di blocchi per raggiungere lo stesso obiettivo, poiché `IDisposable.Dispose` verrà sempre chiamato prima che un'eccezione chiamata all'interno di un blocco `using` inizi a scoppiare nello stack.

Inizializzatori di proprietà automatica

introduzione

Le proprietà possono essere inizializzate con l'operatore `=` dopo la chiusura `}` . La classe `Coordinate` seguito mostra le opzioni disponibili per inizializzare una proprietà:

6.0

```
public class Coordinate
{
    public int X { get; set; } = 34; // get or set auto-property with initializer

    public int Y { get; } = 89;      // read-only auto-property with initializer
}
```

Accessors con diversa visibilità

È possibile inizializzare le proprietà automatiche che hanno visibilità diversa sui loro accessor. Ecco un esempio con un setter protetto:

```
    public string Name { get; protected set; } = "Cheeze";
```

L'accessor può anche essere `internal` , `internal protected` o `private` .

Proprietà di sola lettura

Oltre alla flessibilità con la visibilità, puoi anche inizializzare le proprietà automatiche di sola lettura. Ecco un esempio:

```
    public List<string> Ingredients { get; } =
        new List<string> { "dough", "sauce", "cheese" };
```

Questo esempio mostra anche come inizializzare una proprietà con un tipo complesso. Inoltre, le proprietà automatiche non possono essere solo di scrittura, quindi preclude anche l'inizializzazione di sola scrittura.

Vecchio stile (pre C # 6.0)

Prima del C # 6, questo richiedeva un codice molto più dettagliato. Stavamo usando una variabile extra chiamata backing property per la proprietà per dare un valore predefinito o per inizializzare la proprietà pubblica come di seguito,

6.0

```
public class Coordinate
{
    private int _x = 34;
    public int X { get { return _x; } set { _x = value; } }

    private readonly int _y = 89;
    public int Y { get { return _y; } }

    private readonly int _z;
    public int Z { get { return _z; } }

    public Coordinate()
    {
        _z = 42;
    }
}
```

Nota: *prima di C # 6.0, era ancora possibile inizializzare le* ***proprietà di auto-implementazione di*** *lettura e scrittura (proprietà con getter e setter) all'interno del costruttore, ma non era possibile inizializzare la proprietà in linea con la sua dichiarazione*

Visualizza la demo

---

uso

Gli inizializzatori devono valutare le espressioni statiche, proprio come gli inizializzatori di campo. Se è necessario fare riferimento a membri non statici, è possibile inizializzare le proprietà in costruttori come in precedenza oppure utilizzare proprietà con corpo di espressione. Le espressioni non statiche, come quella seguente (commentate), generano un errore del compilatore:

```
// public decimal X { get; set; } = InitMe();  // generates compiler error

decimal InitMe() { return 4m; }
```

Ma i metodi statici **possono** essere utilizzati per inizializzare le proprietà automatiche:

```
public class Rectangle
{
    public double Length { get; set; } = 1;
    public double Width { get; set; } = 1;
    public double Area { get; set; } = CalculateArea(1, 1);

    public static double CalculateArea(double length, double width)
    {
        return length * width;
    }
}
```

Questo metodo può essere applicato anche a proprietà con diversi livelli di accesso:

```
public short Type { get; private set; } = 15;
```

L'inizializzatore della proprietà automatica consente l'assegnazione di proprietà direttamente all'interno della dichiarazione. Per le proprietà di sola lettura, si prende cura di tutti i requisiti richiesti per garantire che la proprietà sia immutabile. Si consideri, ad esempio, la classe `FingerPrint` nel seguente esempio:

```
public class FingerPrint
{
  public DateTime TimeStamp { get; } = DateTime.UtcNow;

  public string User { get; } =
    System.Security.Principal.WindowsPrincipal.Current.Identity.Name;

  public string Process { get; } =
    System.Diagnostics.Process.GetCurrentProcess().ProcessName;
}
```

Visualizza la demo

Note cautelative

Fai attenzione a non confondere gli inizializzatori di auto-proprietà o di campo con metodi di espressione del corpo simili che utilizzano `=>` anziché `=` , e campi che non includono `{ get; }` .

Ad esempio, ciascuna delle seguenti dichiarazioni è diversa.

```
public class UserGroupDto
{
    // Read-only auto-property with initializer:
    public ICollection<UserDto> Users1 { get; } = new HashSet<UserDto>();

    // Read-write field with initializer:
    public ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // Read-only auto-property with expression body:
    public ICollection<UserDto> Users3 => new HashSet<UserDto>();
}
```

Manca `{ get; }` nella dichiarazione della proprietà risulta in un campo pubblico. Sia gli utenti di proprietà di `Users1` lettura `Users1` che quelli di lettura-scrittura `Users2` vengono inizializzati solo una volta, ma un campo pubblico consente di modificare l'istanza di raccolta dall'esterno della classe, che di solito non è desiderabile. La modifica di una proprietà automatica di sola lettura con il corpo di un'espressione in proprietà di sola lettura con l'inizializzatore richiede non solo la rimozione di `>` da `=>` , ma l'aggiunta di `{ get; }` .

Il diverso simbolo (`=>` invece di `=`) in `Users3` risulta in ogni accesso alla proprietà che restituisce una nuova istanza di `HashSet<UserDto>` che, mentre C # valido (dal punto di vista del compilatore) è improbabile che sia il comportamento desiderato quando usato per un membro della collezione.

Il codice sopra è equivalente a:

```
public class UserGroupDto
{
    // This is a property returning the same instance
    // which was created when the UserGroupDto was instantiated.
    private ICollection<UserDto> _users1 = new HashSet<UserDto>();
    public ICollection<UserDto> Users1 { get { return _users1; } }

    // This is a field returning the same instance
    // which was created when the UserGroupDto was instantiated.
    public virtual ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // This is a property which returns a new HashSet<UserDto> as
    // an ICollection<UserDto> on each call to it.
    public ICollection<UserDto> Users3 { get { return new HashSet<UserDto>(); } }
}
```

Inizializzatori dell'indice

Gli inizializzatori dell'indice consentono di creare e inizializzare oggetti con indici contemporaneamente.

Ciò semplifica l'inizializzazione dei dizionari:

```
var dict = new Dictionary<string, int>()
{
    ["foo"] = 34,
    ["bar"] = 42
};
```

Qualsiasi oggetto che abbia un getter o setter indicizzato può essere usato con questa sintassi:

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42
        };

        Console.ReadKey();
    }
```

```
}
```

Produzione:

Indice: foo, valore: 34
Indice: bar, valore: 42

Visualizza la demo

Se la classe ha più indicizzatori è possibile assegnarli tutti in un singolo gruppo di istruzioni:

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
        public string this[int index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42,
            [10] = "Ten",
            [42] = "Meaning of life"
        };
    }
}
```

Produzione:

Indice: foo, valore: 34
Indice: bar, valore: 42
Indice: 10, valore: dieci
Indice: 42, valore: significato della vita

Va notato che l'accessore del `set` dell'indicizzatore potrebbe comportarsi in modo diverso rispetto a un metodo `Add` (utilizzato negli inizializzatori della raccolta).

Per esempio:

```
var d = new Dictionary<string, int>
{
    ["foo"] = 34,
    ["foo"] = 42,
}; // does not throw, second value overwrites the first one
```

contro:

```
var d = new Dictionary<string, int>
{
    { "foo", 34 },
    { "foo", 42 },
}; // run-time ArgumentException: An item with the same key has already been added.
```

Interpolazione a stringa

L'interpolazione delle stringhe consente allo sviluppatore di combinare `variables` e testo per formare una stringa.

Esempio di base

Vengono create due variabili `int` : `foo` e `bar` .

```
int foo = 34;
int bar = 42;

string resultString = $"The foo is {foo}, and the bar is {bar}.";

Console.WriteLine(resultString);
```

Uscita :

> Il foo è 34 e la barra 42.

Visualizza la demo

Le bretelle all'interno delle stringhe possono ancora essere utilizzate, come questa:

```
var foo = 34;
var bar = 42;

// String interpolation notation (new style)
Console.WriteLine($"The foo is {{foo}}, and the bar is {{bar}}.");
```

Questo produce il seguente risultato:

> Il foo è {pippo} e la barra è {bar}.

Utilizzo dell'interpolazione con letterali stringa letterali

Usando `@` prima della stringa, la stringa verrà interpretata letteralmente. Quindi, ad esempio, i caratteri Unicode o le interruzioni di riga rimarranno esattamente come sono stati digitati. Tuttavia, questo non influirà sulle espressioni in una stringa interpolata come mostrato nell'esempio seguente:

```
Console.WriteLine($@"In case it wasn't clear:
\u00B9
The foo
is {foo},
and the bar
is {bar}.");
```

Produzione:

> Nel caso non fosse chiaro:
> \ u00B9
> Il pippo
> è 34,
> e il bar
> è 42.

Visualizza la demo

---

espressioni

Con l'interpolazione delle stringhe, è possibile valutare anche le *espressioni* all'interno di parentesi graffe `{}` . Il risultato verrà inserito nella posizione corrispondente all'interno della stringa. Ad esempio, per calcolare il massimo di `foo` e `bar` e inserirlo, utilizzare `Math.Max` tra parentesi graffe:

```
Console.WriteLine($"And the greater one is: { Math.Max(foo, bar) }");
```

Produzione:

> E il più grande è: 42

Nota: tutti gli spazi bianchi iniziali o finali (inclusi spazio, scheda e CRLF / nuova riga) tra la parentesi graffa e l'espressione vengono completamente ignorati e non inclusi nell'output

Visualizza la demo

Come altro esempio, le variabili possono essere formattate come valuta:

```
Console.WriteLine($"Foo formatted as a currency to 4 decimal places: {foo:c4}");
```

Produzione:

> Foo formattato come valuta a 4 cifre decimali: $ 34,0000

Visualizza la demo

Oppure possono essere formattati come date:

```
Console.WriteLine($"Today is: {DateTime.Today:dddd, MMMM dd - yyyy}");
```

Produzione:

> Oggi è: lunedì 20 luglio 2015

Visualizza la demo

Le dichiarazioni con un operatore condizionale (Ternario) possono anche essere valutate all'interno dell'interpolazione. Tuttavia, questi devono essere racchiusi tra parentesi, in quanto i due punti vengono utilizzati per indicare la formattazione come mostrato sopra:

```
Console.WriteLine($"{(foo > bar ? "Foo is larger than bar!" : "Bar is larger than foo!")}");
```

Produzione:

> Bar è più grande di foo!

Visualizza la demo

Le espressioni condizionali e gli identificatori di formato possono essere mescolati:

```
Console.WriteLine($"Environment: {(Environment.Is64BitProcess ? 64 : 32):00'-bit'} process");
```

Produzione:

> Ambiente: processo a 32 bit

Sequenze di fuga

L'escape dei caratteri barra rovesciata (`\`) e virgola (`"`) funziona esattamente nello stesso modo nelle stringhe interpolate come nelle stringhe non interpolate, sia per i letterali stringa letterale che per quelli non verbali:

```
Console.WriteLine($"Foo is: {foo}. In a non-verbatim string, we need to escape \" and \\ with
backslashes.");
Console.WriteLine($@"Foo is: {foo}. In a verbatim string, we need to escape "" with an extra
```

```
quote, but we don't need to escape \");
```

Produzione:

> Foo è 34. In una stringa non-letterale, dobbiamo uscire da "e \ con barre retroverse.
> Foo è 34. In una stringa letterale, dobbiamo scappare "con una citazione extra, ma non abbiamo bisogno di scappare \

Per includere una parentesi graffa `{` o `}` in una stringa interpolata, utilizzare due parentesi graffe `{{` o `}}` :

```
$"{{foo}} is: {foo}"
```

Produzione:

> {foo} è: 34

Visualizza la demo

FormattableString type

Il tipo di `$"..."` espressione di interpolazione della stringa non è sempre una stringa semplice. Il compilatore decide quale tipo assegnare a seconda del contesto:

```
string s = $"hello, {name}";
System.FormattableString s = $"Hello, {name}";
System.IFormattable s = $"Hello, {name}";
```

Questo è anche l'ordine delle preferenze di tipo quando il compilatore deve scegliere quale metodo di overload verrà chiamato.

Un nuovo tipo , `System.FormattableString` , rappresenta una stringa di formato composita, insieme agli argomenti da formattare. Usalo per scrivere applicazioni che gestiscono in modo specifico gli argomenti di interpolazione:

```
public void AddLogItem(FormattableString formattableString)
{
    foreach (var arg in formattableString.GetArguments())
    {
        // do something to interpolation argument 'arg'
    }

    // use the standard interpolation and the current culture info
    // to get an ordinary String:
    var formatted = formattableString.ToString();

    // ...
}
```

Chiama il metodo sopra con:

```
AddLogItem($"The foo is {foo}, and the bar is {bar}.");
```

Ad esempio, si potrebbe scegliere di non sostenere il costo delle prestazioni della formattazione della stringa se il livello di registrazione stava già filtrando l'elemento del registro.

Conversioni implicite

Esistono conversioni di tipo implicite da una stringa interpolata:

```
var s = $"Foo: {foo}";
System.IFormattable s = $"Foo: {foo}";
```

Puoi anche produrre una variabile `IFormattable` che ti permette di convertire la stringa con un contesto invariante:

```
var s = $"Bar: {bar}";
System.FormattableString s = $"Bar: {bar}";
```

Metodi di coltura attuali e invariabili

Se l'analisi del codice è attivata, tutte le stringhe interpolate generano l'avviso CA1305 (Specifica `IFormatProvider`). Un metodo statico può essere utilizzato per applicare la cultura corrente.

```
public static class Culture
{
    public static string Current(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.CurrentCulture);
    }
    public static string Invariant(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.InvariantCulture);
    }
}
```

Quindi, per produrre una stringa corretta per la cultura corrente, basta usare l'espressione:

```
Culture.Current($"interpolated {typeof(string).Name} string.")
Culture.Invariant($"interpolated {typeof(string).Name} string.")
```

Nota : `Current` e `Invariant` non possono essere creati come metodi di estensione perché, per impostazione predefinita, il compilatore assegna il tipo `String` *all'espressione di stringa interpolata* che non riesce a compilare il seguente codice:

```
$"interpolated {typeof(string).Name} string.".Current();
```

`FormattableString` classe `FormattableString` contiene già il metodo `Invariant()` , quindi il modo più semplice per passare alla cultura invariante è affidarsi `using static` :

```
using static System.FormattableString;


string invariant = Invariant($"Now = {DateTime.Now}");
string current = $"Now = {DateTime.Now}";
```

Dietro le quinte

Le stringhe interpolate sono solo uno zucchero sintattico per `String.Format()` . Il compilatore (Roslyn) lo trasformerà in un `String.Format` dietro le quinte:

```
var text = $"Hello {name + lastName}";
```

Quanto sopra sarà convertito in qualcosa di simile a questo:

```
string text = string.Format("Hello {0}", new object[] {
    name + lastName
});
```

Interpolazione a stringa e Linq

È possibile utilizzare stringhe interpolate nelle istruzioni Linq per aumentare ulteriormente la leggibilità.

```
var fooBar = (from DataRow x in fooBarTable.Rows
          select string.Format("{0}{1}", x["foo"], x["bar"])).ToList();
```

Può essere riscritto come:

```
var fooBar = (from DataRow x in fooBarTable.Rows
          select $"{x["foo"]}{x["bar"]}").ToList();
```

Stringhe interpolate riutilizzabili

Con `string.Format` , puoi creare stringhe di formato riutilizzabili:

```
public const string ErrorFormat = "Exception caught:\r\n{0}";

// ...
```

```
Logger.Log(string.Format(ErrorFormat, ex));
```

Le stringhe interpolate, tuttavia, non verranno compilate con i segnaposto che fanno riferimento a variabili inesistenti. Quanto segue non verrà compilato:

```
public const string ErrorFormat = $"Exception caught:\r\n{error}";
// CS0103: The name 'error' does not exist in the current context
```

Invece, crea un `Func<>` che consuma variabili e restituisce una `String` :

```
public static Func<Exception, string> FormatError =
    error => $"Exception caught:\r\n{error}";

// ...

Logger.Log(FormatError(ex));
```

Interpolazione e localizzazione delle stringhe

Se stai localizzando la tua applicazione, potresti chiederti se è possibile utilizzare l'interpolazione delle stringhe insieme alla localizzazione. In effetti, sarebbe bello avere la possibilità di memorizzare nel file di risorse `String` s come:

```
"My name is {name} {middlename} {surname}"
```

invece del molto meno leggibile:

```
"My name is {0} {1} {2}"
```

`String` processo di interpolazione delle `String` verifica *in fase di compilazione* , diversamente dalla stringa di formattazione con `string.Format` che si verifica *in fase di runtime* . Le espressioni in una stringa interpolata devono fare riferimento a nomi nel contesto corrente e devono essere archiviate in file di risorse. Ciò significa che se vuoi usare la localizzazione devi farlo come:

```
var FirstName = "John";

// method using different resource file "strings"
// for French ("strings.fr.resx"), German ("strings.de.resx"),
// and English ("strings.en.resx")
void ShowMyNameLocalized(string name, string middlename = "", string surname = "")
{
    // get localized string
    var localizedMyNameIs = Properties.strings.Hello;
    // insert spaces where necessary
    name = (string.IsNullOrWhiteSpace(name) ? "" : name + " ");
    middlename = (string.IsNullOrWhiteSpace(middlename) ? "" : middlename + " ");
    surname = (string.IsNullOrWhiteSpace(surname) ? "" : surname + " ");
    // display it
    Console.WriteLine($"{localizedMyNameIs} {name}{middlename}{surname}".Trim());
}
```

```
// switch to French and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("fr-FR");
ShowMyNameLocalized(FirstName);

// switch to German and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("de-DE");
ShowMyNameLocalized(FirstName);

// switch to US English and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("en-US");
ShowMyNameLocalized(FirstName);
```

Se le stringhe di risorse per le lingue utilizzate in precedenza sono archiviate correttamente nei singoli file di risorse, è necessario ottenere il seguente output:

> Bonjour, mon nom est John
> Ciao, mein Nome ist John
> Ciao il mio nome è John

Nota che ciò implica che il nome segue la stringa localizzata in ogni lingua. In caso contrario, è necessario aggiungere segnaposti alle stringhe di risorse e modificare la funzione in alto oppure è necessario interrogare le informazioni sulla cultura nella funzione e fornire un'istruzione switch case contenente i diversi casi. Per ulteriori dettagli sui file di risorse, vedi Come utilizzare la localizzazione in C # .

È buona norma utilizzare una lingua di fallback predefinita che la maggior parte delle persone capirà, nel caso in cui una traduzione non sia disponibile. Suggerisco di usare l'inglese come lingua di fallback predefinita.

Interpolazione ricorsiva

Sebbene non sia molto utile, è consentito utilizzare una `string` interpolata ricorsivamente all'interno delle parentesi graffe di un'altra:

```
Console.WriteLine($"String has {$"My class is called {nameof(MyClass)}.".Length} chars:");
Console.WriteLine($"My class is called {nameof(MyClass)}.");
```

Produzione:

> La stringa ha 27 caratteri:
>
> La mia classe si chiama MyClass.

Aspettare e prendere

È possibile utilizzare `await` espressione Attendi per applicare l'operatore Attendi a Attività o Attività (di TResult) nel `catch` e `finally` blocchi in C # 6.

Non è stato possibile utilizzare l'espressione `await` nel `catch` e `finally` blocchi nelle versioni

precedenti a causa dei limiti del compilatore. C # 6 rende molto più semplice l'attesa di compiti asincroni consentendo l'espressione di `await` .

```
try
{
    //since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    //since C#6
    await logger.LogAsync(e);
}
finally
{
    //since C#6
    await service.CloseAsync();
}
```

È stato richiesto in C # 5 di usare un `bool` o dichiarare `Exception` al di fuori del try catch per eseguire operazioni asincrone. Questo metodo è mostrato nel seguente esempio:

```
bool error = false;
Exception ex = null;

try
{
    // Since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    // Declare bool or place exception inside variable
    error = true;
    ex = e;
}

// If you don't use the exception
if (error)
{
    // Handle async task
}

// If want to use information from the exception
if (ex != null)
{
    await logger.LogAsync(e);
}

// Close the service, since this isn't possible in the finally
await service.CloseAsync();
```

Propagazione nulla

Il `?.` operatore e operatore `?[...]` sono chiamati operatori condizionali nulli . A volte viene anche indicato con altri nomi come l' operatore di navigazione sicura .

Questo è utile, perché se il `.` L'operatore (accessor membro) viene applicato a un'espressione che restituisce `null` , il programma genererà `NullReferenceException` . Se lo sviluppatore utilizza invece il `?.` (null-condizionale), l'espressione valuterà a null invece di generare un'eccezione.

Si noti che se il `?.` operatore viene utilizzato e l'espressione non è nullo, `?.` e `.` sono equivalenti.

Nozioni di base

```
var teacherName = classroom.GetTeacher().Name;
// throws NullReferenceException if GetTeacher() returns null
```

Visualizza la demo

Se l' `classroom` non ha un insegnante, `GetTeacher()` può restituire `null` . Quando è `null` e si accede alla proprietà `Name` , verrà generata `NullReferenceException` .

Se modifichiamo questa affermazione per usare il `?.` sintassi, il risultato dell'intera espressione sarà `null` :

```
var teacherName = classroom.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null
```

Visualizza la demo

Successivamente, se la `classroom` potrebbe anche essere `null` , potremmo scrivere anche questa affermazione come:

```
var teacherName = classroom?.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null OR classroom is null
```

Visualizza la demo

Questo è un esempio di cortocircuito: quando qualsiasi operazione di accesso condizionale che utilizza l'operatore null-condizionale restituisce null, l'intera espressione restituisce immediatamente null, senza elaborare il resto della catena.

Quando il membro terminale di un'espressione che contiene l'operatore condizionale nullo è di un tipo valore, l'espressione `Nullable<T>` un `Nullable<T>` di quel tipo e quindi non può essere utilizzato come sostituzione diretta dell'espressione senza `?.` .

```
bool hasCertification = classroom.GetTeacher().HasCertification;
// compiles without error but may throw a NullReferenceException at runtime

bool hasCertification = classroom?.GetTeacher()?.HasCertification;
// compile time error: implicit conversion from bool? to bool not allowed

bool? hasCertification = classroom?.GetTeacher()?.HasCertification;
// works just fine, hasCertification will be null if any part of the chain is null
```

```
bool hasCertification = classroom?.GetTeacher()?.HasCertification.GetValueOrDefault();
// must extract value from nullable to assign to a value type variable
```

Utilizzare con l'operatore Null-Coalescing (??)

È possibile combinare l'operatore null-condizionale con l'operatore Null-coalescing (`??`) per restituire un valore predefinito se l'espressione si risolve in `null` . Usando il nostro esempio sopra:

```
var teacherName = classroom?.GetTeacher()?.Name ?? "No Name";
// teacherName will be "No Name" when GetTeacher()
// returns null OR classroom is null OR Name is null
```

Utilizzare con gli indicizzatori

L'operatore null-condizionale può essere utilizzato con gli indicizzatori :

```
var firstStudentName = classroom?.Students?[0]?.Name;
```

Nell'esempio sopra:

- Il primo `?.` assicura che l' `classroom` non sia `null` .
- Il secondo `?` assicura che l'intera raccolta `Students` non sia `null` .
- Il terzo `?.` dopo che l'indicizzatore assicura che l'indicizzatore `[0]` non ha restituito un oggetto `null` . Va notato che questa operazione può **ancora** lanciare un `IndexOutOfRangeException` .

Utilizzare con funzioni void

L'operatore Null-condition può anche essere utilizzato con le funzioni `void` . Tuttavia, in questo caso, la dichiarazione non valuterà `null` . `NullReferenceException` solo una `NullReferenceException` .

```
List<string> list = null;
list?.Add("hi");          // Does not evaluate to null
```

Utilizzare con invocazione di eventi

Supponendo la seguente definizione di evento:

```
private event EventArgs OnCompleted;
```

Quando si invoca un evento, tradizionalmente, è consigliabile verificare se l'evento è `null` nel caso in cui non siano presenti abbonati:

```
var handler = OnCompleted;
if (handler != null)
{
    handler(EventArgs.Empty);
}
```

Poiché è stato introdotto l'operatore null-condizionale, l'invocazione può essere ridotta a una singola riga:

```
OnCompleted?.Invoke(EventArgs.Empty);
```

---

limitazioni

L'operatore Null-condition produce rvalue, non lvalue, cioè non può essere utilizzato per l'assegnazione di proprietà, la sottoscrizione di eventi ecc. Ad esempio, il seguente codice non funzionerà:

```
// Error: The left-hand side of an assignment must be a variable, property or indexer
Process.GetProcessById(1337)?.EnableRaisingEvents = true;
// Error: The event can only appear on the left hand side of += or -=
Process.GetProcessById(1337)?.Exited += OnProcessExited;
```

---

gotchas

Nota che:

```
int? nameLength = person?.Name.Length;    // safe if 'person' is null
```

non è la stessa di:

```
int? nameLength = (person?.Name).Length;  // avoid this
```

perché il primo corrisponde a:

```
int? nameLength = person != null ? (int?)person.Name.Length : null;
```

e quest'ultimo corrisponde a:

```
int? nameLength = (person != null ? person.Name : null).Length;
```

Nonostante l'operatore ternario `?:` Qui viene utilizzato per spiegare la differenza tra due casi, questi operatori non sono equivalenti. Questo può essere facilmente dimostrato con il seguente esempio:

```
void Main()
{
    var foo = new Foo();
    Console.WriteLine("Null propagation");
    Console.WriteLine(foo.Bar?.Length);

    Console.WriteLine("Ternary");
    Console.WriteLine(foo.Bar != null ? foo.Bar.Length : (int?)null);
}

class Foo
{
    public string Bar
    {
        get
        {
            Console.WriteLine("I was read");
            return string.Empty;
        }
    }
}
```

Quali uscite:

Propagazione nulla
Sono stato letto
0
Ternario
Sono stato letto
Sono stato letto
0

Visualizza la demo

Per evitare invocazioni multiple equivalenti sarebbe:

```
var interimResult = foo.Bar;
Console.WriteLine(interimResult != null ? interimResult.Length : (int?)null);
```

E questa differenza spiega in qualche modo perché l'operatore di propagazione null non è ancora supportato negli alberi di espressione.

Uso del tipo statico

L' `using static [Namespace.Type]` direttiva `using static [Namespace.Type]` consente l'importazione di membri statici di tipi e valori di enumerazione. I metodi di estensione vengono importati come metodi di estensione (da un solo tipo), non in ambito di livello superiore.

6.0

```
using static System.Console;
using static System.ConsoleColor;
using static System.Math;

class Program
{
    static void Main()
    {
        BackgroundColor = DarkBlue;
        WriteLine(Sqrt(2));
    }
}
```

Live Demo Fiddle

6.0

```
using System;

class Program
{
    static void Main()
    {
        Console.BackgroundColor = ConsoleColor.DarkBlue;
        Console.WriteLine(Math.Sqrt(2));
    }
}
```

Migliore risoluzione del sovraccarico

Lo snippet seguente mostra un esempio di passaggio di un gruppo di metodi (al contrario di un lambda) quando è previsto un delegato. Risoluzione di sovraccarico ora risolverà questo invece di sollevare un ambiguo errore di sovraccarico dovuto alla capacità di **C # 6** di controllare il tipo di ritorno del metodo che è stato passato.

```
using System;
public class Program
{
    public static void Main()
    {
        Overloaded(DoSomething);
    }

    static void Overloaded(Action action)
    {
       Console.WriteLine("overload with action called");
    }

    static void Overloaded(Func<int> function)
    {
       Console.WriteLine("overload with Func<int> called");
    }

    static int DoSomething()
    {
        Console.WriteLine(0);
        return 0;
```

```
    }
}
```

risultati:

6.0

Produzione

> sovraccarico con Func <int> chiamato

Visualizza la demo

5.0

Errore

> errore CS0121: la chiamata è ambigua tra i seguenti metodi o proprietà: "Program.Overloaded (System.Action)" e "Program.Overloaded (System.Func)"

C # 6 può anche gestire bene il seguente caso di corrispondenza esatta per espressioni lambda che avrebbe comportato un errore in **C # 5** .

```
using System;

class Program
{
    static void Foo(Func<Func<long>> func) {}
    static void Foo(Func<Func<int>> func) {}

    static void Main()
    {
        Foo(() => () => 7);
    }
}
```

Cambiamenti minori e correzioni di errori

Le parentesi ora sono vietate attorno ai parametri denominati. Il seguente compila in C # 5, ma non C # 6

5.0

```
Console.WriteLine((value: 23));
```

L'operando di `is` e `as` non è più possibile essere gruppi di metodi. Il seguente compila in C # 5, ma non C # 6

5.0

```
var result = "".Any is byte;
```

> Il compilatore nativo ha permesso questo (anche se ha mostrato un avvertimento), e in effetti non ha nemmeno controllato la compatibilità del metodo di estensione, permettendo cose pazzesche come `1.Any is string` o `IDisposable.Dispose is object` .

Vedi questo riferimento per gli aggiornamenti sulle modifiche.

Utilizzo di un metodo di estensione per l'inizializzazione della raccolta

La sintassi di inizializzazione della raccolta può essere utilizzata durante l'istanziazione di qualsiasi classe che implementa `IEnumerable` e ha un metodo denominato `Add` che accetta un singolo parametro.

Nelle versioni precedenti, questo metodo `Add` doveva essere un metodo di **istanza** sulla classe da inizializzare. In C # 6, può anche essere un metodo di estensione.

```
public class CollectionWithAdd : IEnumerable
{
    public void Add<T>(T item)
    {
        Console.WriteLine("Item added with instance add method: " + item);
    }

    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public class CollectionWithoutAdd : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public static class Extensions
{
    public static void Add<T>(this CollectionWithoutAdd collection, T item)
    {
        Console.WriteLine("Item added with extension add method: " + item);
    }
}

public class Program
{
    public static void Main()
    {
        var collection1 = new CollectionWithAdd{1,2,3}; // Valid in all C# versions
        var collection2 = new CollectionWithoutAdd{4,5,6}; // Valid only since C# 6
    }
}
```

Questo produrrà:

> Elemento aggiunto con metodo di aggiunta istanza: 1

Elemento aggiunto con metodo di aggiunta istanza: 2
Elemento aggiunto con metodo di aggiunta istanza: 3
Elemento aggiunto con metodo di aggiunta estensione: 4
Elemento aggiunto con il metodo di aggiunta estensione: 5
Elemento aggiunto con il metodo di aggiunta estensione: 6

Disabilita i miglioramenti degli avvisi

In C # 5.0 e precedenti lo sviluppatore poteva solo sopprimere gli avvertimenti per numero. Con l'introduzione di Roslyn Analyzer, C # ha bisogno di un modo per disabilitare gli avvisi emessi da librerie specifiche. Con C # 6.0 la direttiva pragma può sopprimere gli avvertimenti per nome.

Prima:

```
#pragma warning disable 0501
```

C # 6.0:

```
#pragma warning disable CS0501
```

Leggi C # 6.0 Caratteristiche online: https://riptutorial.com/it/csharp/topic/24/c-sharp-6-0-caratteristiche

Capitolo 21: C # 7.0 Caratteristiche

introduzione

C # 7.0 è la settima versione di C #. Questa versione contiene alcune nuove funzionalità: supporto linguistico per Tuple, funzioni locali, dichiarazioni `out var` , separatori di cifre, valori letterali binari, corrispondenza di pattern, espressioni di lancio, `ref return` e `ref local` list di membri body con espressioni `ref local` ed estese.

Riferimento ufficiale: Novità in C # 7

Examples

out var dichiarazione

Un modello comune in C # sta utilizzando `bool TryParse(object input, out object value)` per analizzare in modo sicuro gli oggetti.

La dichiarazione `out var` è una funzione semplice per migliorare la leggibilità. Permette di dichiarare una variabile nello stesso momento in cui viene passata come parametro out.

Una variabile dichiarata in questo modo è portata al resto del corpo nel punto in cui è dichiarata.

Esempio

Usando `TryParse` prima di C # 7.0, devi dichiarare una variabile per ricevere il valore prima di chiamare la funzione:

7.0

```
int value;
if (int.TryParse(input, out value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // ok
```

In C # 7.0, è possibile allineare la dichiarazione della variabile passata al parametro `out` , eliminando la necessità di una dichiarazione di variabile separata:

7.0

```
if (int.TryParse(input, out var value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // still ok, the value in scope within the remainder of the body
```

Se alcuni dei parametri che una funzione ritorna in `out` non è necessario, è possibile utilizzare l'operatore di *scarto* `_` .

```
p.GetCoordinates(out var x, out _); // I only care about x
```

Una dichiarazione `out var` può essere utilizzata con qualsiasi funzione esistente che ha già parametri `out` . La sintassi della dichiarazione di funzione rimane la stessa e non sono necessari requisiti aggiuntivi per rendere la funzione compatibile con una dichiarazione `out var` . Questa caratteristica è semplicemente zucchero sintattico.

Un'altra caratteristica della dichiarazione `out var` è che può essere utilizzato con tipi anonimi.

7.0

```
var a = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var groupedByMod2 = a.Select(x => new
                                  {
                                      Source = x,
                                      Mod2 = x % 2
                                  })
                     .GroupBy(x => x.Mod2)
                     .ToDictionary(g => g.Key, g => g.ToArray());
if (groupedByMod2.TryGetValue(1, out var oddElements))
{
    Console.WriteLine(oddElements.Length);
}
```

In questo codice creiamo un `Dictionary` con chiave `int` e array di valore di tipo anonimo. Nella versione precedente di C # era impossibile utilizzare qui il metodo `TryGetValue` poiché richiedeva di dichiarare la variabile `out` (che è di tipo anonimo!). Tuttavia, con `out var` non è necessario specificare esplicitamente il tipo di variabile `out` .

limitazioni

Si noti che le dichiarazioni var out sono di uso limitato nelle query LINQ poiché le espressioni vengono interpretate come corpi lambda espressione, quindi l'ambito delle variabili introdotte è limitato a questi lambda. Ad esempio, il seguente codice non funzionerà:

```
var nums =
    from item in seq
```

```
    let success = int.TryParse(item, out var tmp)
    select success ? tmp : 0; // Error: The name 'tmp' does not exist in the current context
```

Riferimenti

- Proposta di dichiarazione originale out var su GitHub

Letterali binari

Il prefisso **0b** può essere utilizzato per rappresentare i valori letterali binari.

I letterali binari consentono di costruire numeri da zero e uno, il che rende molto più facile vedere quali bit sono impostati nella rappresentazione binaria di un numero. Questo può essere utile per lavorare con i flag binari.

I seguenti sono modi equivalenti per specificare un valore `int` con il valore `34` (= $2^5 + 2^1$):

```
// Using a binary literal:
//   bits: 76543210
int a1 = 0b00100010;          // binary: explicitly specify bits

// Existing methods:
int a2 = 0x22;                // hexadecimal: every digit corresponds to 4 bits
int a3 = 34;                  // decimal: hard to visualise which bits are set
int a4 = (1 << 5) | (1 << 1); // bitwise arithmetic: combining non-zero bits
```

Elenchi di bandiere

Prima, specificare i valori dei flag per un `enum` poteva essere fatto usando uno dei tre metodi in questo esempio:

```
[Flags]
public enum DaysOfWeek
{
    // Previously available methods:
    //          decimal        hex       bit shifting
    Monday    =  1,    //    = 0x01    = 1 << 0
    Tuesday   =  2,    //    = 0x02    = 1 << 1
    Wednesday =  4,    //    = 0x04    = 1 << 2
    Thursday  =  8,    //    = 0x08    = 1 << 3
    Friday    = 16,    //    = 0x10    = 1 << 4
    Saturday  = 32,    //    = 0x20    = 1 << 5
    Sunday    = 64,    //    = 0x40    = 1 << 6

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}
```

Con i letterali binari è più ovvio quali bit sono impostati e il loro utilizzo non richiede la comprensione di numeri esadecimali e aritmetica bit a bit:

```
[Flags]
public enum DaysOfWeek
{
    Monday    = 0b00000001,
    Tuesday   = 0b00000010,
    Wednesday = 0b00000100,
    Thursday  = 0b00001000,
    Friday    = 0b00010000,
    Saturday  = 0b00100000,
    Sunday    = 0b01000000,

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}
```

Separatori di cifre

Il carattere di sottolineatura _ può essere utilizzato come separatore di cifre. Essere in grado di raggruppare cifre in grandi valori letterali numerici ha un impatto significativo sulla leggibilità.

Il carattere di sottolineatura può comparire ovunque in un valore letterale numerico, ad eccezione di quanto indicato di seguito. Raggruppamenti diversi possono avere senso in diversi scenari o con differenti basi numeriche.

Qualsiasi sequenza di cifre può essere separata da uno o più caratteri di sottolineatura. Il _ è permesso in decimali così come in esponenti. I separatori non hanno alcun impatto semantico: sono semplicemente ignorati.

```
int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2__3___4____5_____6______7_______8________9;
double real = 1_000.111_1e-1_000;
```

Dove non è possibile utilizzare il separatore di cifre _ :

- all'inizio del valore (`_121`)
- alla fine del valore (`121_` o `121.05_`)
- accanto al decimale (`10_.0`)
- accanto al carattere esponente (`1.1e_1`)
- accanto allo specificatore di tipo (`10_f`)
- immediatamente dopo il `0x` o `0b` in letterali binari ed esadecimali (potrebbe essere modificato per consentire, ad esempio, 0b_1001_1000)

Supporto linguistico per Tuples

Nozioni di base

Una **tupla** è un elenco di elementi ordinato e finito. Le tuple sono comunemente utilizzate nella programmazione come mezzo per lavorare collettivamente con una singola entità invece di

lavorare individualmente con ciascuno degli elementi della tupla e per rappresentare singole righe (cioè "record") in un database relazionale.

In C # 7.0, i metodi possono avere più valori di ritorno. Dietro le quinte, il compilatore utilizzerà la nuova struttura ValueTuple .

```
public (int sum, int count) GetTallies()
{
    return (1, 2);
}
```

Nota a margine : per farlo funzionare in Visual Studio 2017, è necessario ottenere il pacchetto `System.ValueTuple` .

Se un risultato del metodo tuple-return è assegnato a una singola variabile, è possibile accedere ai membri con i loro nomi definiti sulla firma del metodo:

```
var result = GetTallies();
// > result.sum
// 1
// > result.count
// 2
```

Decuplicazione delle tuple

La decostruzione della tupla separa una tupla nelle sue parti.

Ad esempio, il `GetTallies` e l'assegnazione del valore di ritorno a due variabili separate decostruiscono la tupla in queste due variabili:

```
(int tallyOne, int tallyTwo) = GetTallies();
```

`var` funziona anche:

```
(var s, var c) = GetTallies();
```

Puoi anche usare la sintassi più breve, con `var` al di fuori di `()` :

```
var (s, c) = GetTallies();
```

Puoi anche decostruire in variabili esistenti:

```
int s, c;
(s, c) = GetTallies();
```

Lo swapping ora è molto più semplice (non è necessaria alcuna variabile temporanea):

```
(b, a) = (a, b);
```

È interessante notare che qualsiasi oggetto può essere decostruito definendo un metodo `Deconstruct` nella classe:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}

var person = new Person { FirstName = "John", LastName = "Smith" };
var (localFirstName, localLastName) = person;
```

In questo caso, la sintassi della `(localFirstName, localLastName) = person` sta invocando `Deconstruct` sulla `person` .

La decostruzione può anche essere definita in un metodo di estensione. Questo è equivalente a quanto sopra:

```
public static class PersonExtensions
{
    public static void Deconstruct(this Person person, out string firstName, out string
lastName)
    {
        firstName = person.FirstName;
        lastName = person.LastName;
    }
}

var (localFirstName, localLastName) = person;
```

Un approccio alternativo per la classe `Person` è definire il `Name` stesso come una `Tuple` . Considera quanto segue:

```
class Person
{
    public (string First, string Last) Name { get; }

    public Person((string FirstName, string LastName) name)
    {
        Name = name;
    }
}
```

Quindi puoi istanziare una persona in questo modo (dove possiamo prendere una tupla come argomento):

```
var person = new Person(("Jane", "Smith"));

var firstName = person.Name.First; // "Jane"
```

```
var lastName = person.Name.Last;   // "Smith"
```

Inizializzazione della tupla

Puoi anche creare arbitrariamente tuple nel codice:

```
var name = ("John", "Smith");
Console.WriteLine(name.Item1);
// Outputs John

Console.WriteLine(name.Item2);
// Outputs Smith
```

Quando si crea una tupla, è possibile assegnare nomi di elementi ad hoc ai membri della tupla:

```
var name = (first: "John", middle: "Q", last: "Smith");
Console.WriteLine(name.first);
// Outputs John
```

Tipo di inferenza

Più tuple definite con la stessa firma (tipi e conteggi corrispondenti) saranno dedotti come tipi corrispondenti. Per esempio:

```
public (int sum, double average) Measure(List<int> items)
{
    var stats = (sum: 0, average: 0d);
    stats.sum = items.Sum();
    stats.average = items.Average();
    return stats;
}
```

`stats` possono essere restituite poiché la dichiarazione della variabile `stats` e la firma di ritorno del metodo sono una corrispondenza.

Nomi di campo di riflessione e tupla

I nomi dei membri non esistono in fase di runtime. Reflection considererà le tuple con lo stesso numero e tipi di membri uguali anche se i nomi dei membri non corrispondono. Convertire una tupla in un `object` e poi in una tupla con gli stessi tipi di membri, ma nomi diversi, non causerà nemmeno un'eccezione.

Mentre la classe ValueTuple stessa non conserva le informazioni per i nomi dei membri, le informazioni sono disponibili tramite la riflessione in TupleElementNamesAttribute. Questo attributo non viene applicato alla tupla stessa ma ai parametri del metodo, ai valori restituiti, alle

proprietà e ai campi. Ciò consente di preservare i nomi delle tuple attraverso gli assembly, ovvero se un metodo restituisce (nome stringa, conteggio int) il nome e il conteggio dei nomi saranno disponibili ai chiamanti del metodo in un altro assembly poiché il valore restituito sarà contrassegnato con TupleElementNameAttribute che contiene i valori "nome" e "conteggio".

Utilizzare con generici e `async`

Le nuove funzionalità di tuple (che utilizzano il tipo `ValueTuple` sottostante) supportano completamente i generici e possono essere utilizzate come parametri di tipo generico. Ciò rende possibile usarli con il modello `async` / `await` :

```
public async Task<(string value, int count)> GetValueAsync()
{
    string fooBar = await _stackoverflow.GetStringAsync();
    int num = await _stackoverflow.GetIntAsync();

    return (fooBar, num);
}
```

Utilizzare con le raccolte

Potrebbe essere utile disporre di una raccolta di tuple (ad esempio) in uno scenario in cui si stia tentando di trovare una tupla corrispondente in base alle condizioni per evitare la ramificazione del codice.

Esempio:

```
private readonly List<Tuple<string, string, string>> labels = new List<Tuple<string, string,
string>>()
{
    new Tuple<string, string, string>("test1", "test2", "Value"),
    new Tuple<string, string, string>("test1", "test1", "Value2"),
    new Tuple<string, string, string>("test2", "test2", "Value3"),
};

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.Item1 == firstElement && w.Item2 == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.Item3;
}
```

Con le nuove tuple può diventare:

```
private readonly List<(string firstThingy, string secondThingyLabel, string foundValue)>
```

```
labels = new List<(string firstThingy, string secondThingyLabel, string foundValue)>()
{
    ("test1", "test2", "Value"),
    ("test1", "test1", "Value2"),
    ("test2", "test2", "Value3"),
}

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.firstThingy == firstElement && w.secondThingyLabel == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.foundValue;
}
```

Sebbene la denominazione sopra la tupla di esempio sopra sia piuttosto generica, l'idea di etichette pertinenti consente una comprensione più profonda di ciò che viene tentato nel codice rispetto a "item1", "item2" e "item3".

Differenze tra ValueTuple e Tuple

La ragione principale per l'introduzione di `ValueTuple` è la prestazione.

Digita il nome	`ValueTuple`	`Tuple`
Classe o struttura	`struct`	`class`
Mutabilità (modifica dei valori dopo la creazione)	mutevole	immutabile
Assegnazione di nomi ai membri e supporto per altre lingue	sì	no (TBD)

Riferimenti

- Proposta di caratteristiche linguistiche originali Tuples su GitHub
- Una soluzione VS 15 percorribile per le funzionalità C # 7.0
- NuGet Tuple Package

Funzioni locali

Le funzioni locali sono definite all'interno di un metodo e non sono disponibili al di fuori di esso. Hanno accesso a tutte le variabili locali e supportano iteratori, `async` / `await` e sintassi lambda. In questo modo, le ripetizioni specifiche di una funzione possono essere funzionalizzate senza affollare la classe. Come effetto collaterale, questo migliora le prestazioni di suggerimento intellisense.

Esempio

```
double GetCylinderVolume(double radius, double height)
{
    return getVolume();

    double getVolume()
    {
        // You can declare inner-local functions in a local function
        double GetCircleArea(double r) => Math.PI * r * r;

        // ALL parents' variables are accessible even though parent doesn't have any input.
        return GetCircleArea(radius) * height;
    }
}
```

Le funzioni locali semplificano considerevolmente il codice per gli operatori LINQ, in cui di solito è necessario separare i controlli degli argomenti dalla logica effettiva per rendere istantanei gli assegni degli argomenti, non ritardati fino a dopo l'avvio dell'iterazione.

Esempio

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null) throw new ArgumentNullException(nameof(source));
    if (predicate == null) throw new ArgumentNullException(nameof(predicate));

    return iterator();

    IEnumerable<TSource> iterator()
    {
        foreach (TSource element in source)
            if (predicate(element))
                yield return element;
    }
}
```

Le funzioni locali supportano anche le parole chiave `async` e `await` .

Esempio

```
async Task WriteEmailsAsync()
{
    var emailRegex = new Regex(@"(?i)[a-z0-9_.+-]+@[a-z0-9-]+\.[a-z0-9-.]+");
    IEnumerable<string> emails1 = await getEmailsFromFileAsync("input1.txt");
    IEnumerable<string> emails2 = await getEmailsFromFileAsync("input2.txt");
    await writeLinesToFileAsync(emails1.Concat(emails2), "output.txt");

    async Task<IEnumerable<string>> getEmailsFromFileAsync(string fileName)
```

```
    {
        string text;

        using (StreamReader reader = File.OpenText(fileName))
        {
            text = await reader.ReadToEndAsync();
        }

        return from Match emailMatch in emailRegex.Matches(text) select emailMatch.Value;
    }

    async Task writeLinesToFileAsync(IEnumerable<string> lines, string fileName)
    {
        using (StreamWriter writer = File.CreateText(fileName))
        {
            foreach (string line in lines)
            {
                await writer.WriteLineAsync(line);
            }
        }
    }
}
```

Una cosa importante che potresti aver notato è che le funzioni locali possono essere definite sotto l'istruzione `return` , **non è** necessario che siano definite sopra di essa. Inoltre, le funzioni locali in genere seguono la convenzione di denominazione "lowerCamelCase" per distinguersi più facilmente dalle funzioni dell'ambito di classe.

Pattern Matching

Le estensioni di pattern matching per C # abilitano molti dei vantaggi dell'abbinamento di pattern da linguaggi funzionali, ma in un modo che si integra perfettamente con la sensazione del linguaggio sottostante

`switch` espressione

Pattern matching estende l' `switch` dichiarazione per accendere i tipi:

```
class Geometry {}

class Triangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
    public int Base { get; set; }
}

class Rectangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
}

class Square : Geometry
{
```

```
    public int Width { get; set; }
}

public static void PatternMatching()
{
    Geometry g = new Square { Width = 5 };

    switch (g)
    {
        case Triangle t:
            Console.WriteLine($"{t.Width} {t.Height} {t.Base}");
            break;
        case Rectangle sq when sq.Width == sq.Height:
            Console.WriteLine($"Square rectangle: {sq.Width} {sq.Height}");
            break;
        case Rectangle r:
            Console.WriteLine($"{r.Width} {r.Height}");
            break;
        case Square s:
            Console.WriteLine($"{s.Width}");
            break;
        default:
            Console.WriteLine("<other>");
            break;
    }
}
```

`is` espressione

La corrispondenza del modello estende l'operatore `is` per controllare un tipo e dichiarare una nuova variabile allo stesso tempo.

Esempio

7.0

```
string s = o as string;
if(s != null)
{
    // do something with s
}
```

può essere riscritto come:

7.0

```
if(o is string s)
{
    //Do something with s
};
```

Si noti inoltre che l'ambito della variabile di pattern `s` viene esteso al di fuori del blocco `if` che raggiunge la fine dell'ambito di inclusione, ad esempio:

```
if(someCondition)
{
   if(o is string s)
   {
      //Do something with s
   }
   else
   {
     // s is unassigned here, but accessible
   }

   // s is unassigned here, but accessible
}
// s is not accessible here
```

ref ritorno e ref locale

Ref return e ref locals sono utili per manipolare e restituire riferimenti a blocchi di memoria invece di copiare memoria senza ricorrere a puntatori non sicuri.

Restituzione

```
public static ref TValue Choose<TValue>(
    Func<bool> condition, ref TValue left, ref TValue right)
{
    return condition() ? ref left : ref right;
}
```

Con questo puoi passare due valori per riferimento con uno di essi restituito in base ad alcune condizioni:

```
Matrix3D left = …, right = …;
Choose(chooser, ref left, ref right).M20 = 1.0;
```

Ref locale

```
public static ref int Max(ref int first, ref int second, ref int third)
{
    ref int max = first > second ? ref first : ref second;
    return max > third ? ref max : ref third;
}
…
int a = 1, b = 2, c = 3;
Max(ref a, ref b, ref c) = 4;
Debug.Assert(a == 1); // true
Debug.Assert(b == 2); // true
Debug.Assert(c == 4); // true
```

Operazioni di riferimento non sicure

In `System.Runtime.CompilerServices.Unsafe` una serie di operazioni non sicure sono stati definiti che permettono di manipolare `ref` valori come se fossero puntatori, in fondo.

Ad esempio, reinterpretare un indirizzo di memoria (`ref`) come un tipo diverso:

```
byte[] b = new byte[4] { 0x42, 0x42, 0x42, 0x42 };

ref int r = ref Unsafe.As<byte, int>(ref b[0]);
Assert.Equal(0x42424242, r);

0x0EF00EF0;
Assert.Equal(0xFE, b[0] | b[1] | b[2] | b[3]);
```

Attenzione però a fare questo, ad esempio, controlla `BitConverter.IsLittleEndian` se necessario e `BitConverter.IsLittleEndian` conseguenza.

Oppure scorrere su un array in modo non sicuro:

```
int[] a = new int[] { 0x123, 0x234, 0x345, 0x456 };

ref int r1 = ref Unsafe.Add(ref a[0], 1);
Assert.Equal(0x234, r1);

ref int r2 = ref Unsafe.Add(ref r1, 2);
Assert.Equal(0x456, r2);

ref int r3 = ref Unsafe.Add(ref r2, -3);
Assert.Equal(0x123, r3);
```

O la `Subtract` simile:

```
string[] a = new string[] { "abc", "def", "ghi", "jkl" };

ref string r1 = ref Unsafe.Subtract(ref a[0], -2);
Assert.Equal("ghi", r1);

ref string r2 = ref Unsafe.Subtract(ref r1, -1);
Assert.Equal("jkl", r2);

ref string r3 = ref Unsafe.Subtract(ref r2, 3);
Assert.Equal("abc", r3);
```

Inoltre, si può controllare se due `ref` valori sono uguali stesso indirizzo cioè:

```
long[] a = new long[2];

Assert.True(Unsafe.AreSame(ref a[0], ref a[0]));
Assert.False(Unsafe.AreSame(ref a[0], ref a[1]));
```

link

Roslyn Github Issue

System.Runtime.CompilerServices.Unsafe su github

lanciare espressioni

C # 7.0 consente di lanciare come espressione in determinati punti:

```
class Person
{
    public string Name { get; }

    public Person(string name) => Name = name ?? throw new
ArgumentNullException(nameof(name));

    public string GetFirstName()
    {
        var parts = Name.Split(' ');
        return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No
name!");
    }

    public string GetLastName() => throw new NotImplementedException();
}
```

Prima di C # 7.0, se si desidera generare un'eccezione da un corpo di espressione, è necessario:

```
var spoons = "dinner,desert,soup".Split(',');

var spoonsArray = spoons.Length > 0 ? spoons : null;

if (spoonsArray == null)
{
    throw new Exception("There are no spoons");
}
```

O

```
var spoonsArray = spoons.Length > 0
    ? spoons
    : new Func<string[]>(() =>
      {
          throw new Exception("There are no spoons");
      })();
```

In C # 7.0 quanto sopra è ora semplificato per:

```
var spoonsArray = spoons.Length > 0 ? spoons : throw new Exception("There are no spoons");
```

Elenco dei membri del corpo con espressione estesa

C # 7.0 aggiunge accessor, costruttori e finalizzatori all'elenco di cose che possono avere corpi di espressioni:

```
class Person
```

```
{
    private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int,
string>();

    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name); // constructors

    ~Person() => names.TryRemove(id, out _);              // finalizers

    public string Name
    {
        get => names[id];                                 // getters
        set => names[id] = value;                         // setters
    }
}
```

Vedere anche la sezione di dichiarazione out var per l'operatore di scarto.

ValueTask

`Task<T>` è una **classe** e causa l'inutile sovraccarico della sua allocazione quando il risultato è immediatamente disponibile.

`ValueTask<T>` è una **struttura** ed è stato introdotto per impedire l'allocazione di un oggetto `Task` nel caso in cui il risultato dell'operazione **asincrona** sia già disponibile al momento dell'attesa.

Quindi `ValueTask<T>` offre due vantaggi:

1. Aumento delle prestazioni

Ecco un esempio di `Task<T>` :

- Richiede allocazione dell'heap
- Prende 120ns con JIT

```
async Task<int> TestTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

Ecco l'esempio analogico `ValueTask<T>` :

- No allocazione heap se il risultato è noto in modo sincrono (il che non è in questo caso a causa della `Task.Delay` , ma spesso è in molti nel mondo reale `async` / `await` scenari)
- Prende 65ns con JIT

```
async ValueTask<int> TestValueTask(int d)
{
    await Task.Delay(d);
    return 10;
```

```
}
```

2. Maggiore flessibilità di implementazione

Le implementazioni di un'interfaccia asincrona che desiderano essere sincrone sarebbero altrimenti costrette a utilizzare `Task.Run` o `Task.FromResult` (con conseguente penalizzazione delle prestazioni discussa sopra). Quindi c'è una certa pressione contro le implementazioni sincrone.

Ma con `ValueTask<T>` , le implementazioni sono più libere di scegliere tra l'essere sincrono o asincrono senza impatto sui chiamanti.

Ad esempio, ecco un'interfaccia con un metodo asincrono:

```
interface IFoo<T>
{
    ValueTask<T> BarAsync();
}
```

... ed ecco come si potrebbe chiamare quel metodo:

```
IFoo<T> thing = getThing();
var x = await thing.BarAsync();
```

Con `ValueTask` , il codice sopra funzionerà con **implementazioni sincrone o asincrone** :

Implementazione sincrona:

```
class SynchronousFoo<T> : IFoo<T>
{
    public ValueTask<T> BarAsync()
    {
        var value = default(T);
        return new ValueTask<T>(value);
    }
}
```

Implementazione asincrona

```
class AsynchronousFoo<T> : IFoo<T>
{
    public async ValueTask<T> BarAsync()
    {
        var value = default(T);
        await Task.Delay(1);
        return value;
    }
}
```

Gli appunti

Sebbene la struttura `ValueTask` fosse stata pianificata per essere aggiunta a C # 7.0 , per ora è stata conservata come un'altra libreria. ValueTask <T> Il pacchetto `System.Threading.Tasks.Extensions` può essere scaricato dalla Galleria Nuget

Leggi C # 7.0 Caratteristiche online: https://riptutorial.com/it/csharp/topic/1936/c-sharp-7-0-caratteristiche

Capitolo 22: C # Script

Examples

Valutazione del codice semplice

Puoi valutare qualsiasi codice C # valido:

```
int value = await CSharpScript.EvaluateAsync<int>("15 * 89 + 95");
var span = await CSharpScript.EvaluateAsync<TimeSpan>("new DateTime(2016,1,1) -
DateTime.Now");
```

Se type non è specificato, il risultato è `object` :

```
object value = await CSharpScript.EvaluateAsync("15 * 89 + 95");
```

Leggi C # Script online: https://riptutorial.com/it/csharp/topic/3780/c-sharp-script

Capitolo 23: caching

Examples

MemoryCache

```
//Get instance of cache
using System.Runtime.Caching;

var cache = MemoryCache.Default;

//Check if cache contains an item with
cache.Contains("CacheKey");

//get item from cache
var item = cache.Get("CacheKey");

//get item from cache or add item if not existing
object list = MemoryCache.Default.AddOrGetExisting("CacheKey", "object to be stored",
DateTime.Now.AddHours(12));

//note if item not existing the item is added by this method
//but the method returns null
```

Leggi caching online: https://riptutorial.com/it/csharp/topic/4383/caching

Capitolo 24: Classe e metodi parziali

introduzione

Le classi parziali ci forniscono un'opzione per dividere le classi in più parti e in più file sorgente. Tutte le parti sono combinate in una singola classe durante la compilazione. Tutte le parti dovrebbero contenere la parola chiave `partial` , dovrebbero essere della stessa accessibilità. Tutte le parti devono essere presenti nello stesso assieme per essere incluse durante la compilazione.

Sintassi

- public **partial** class MyPartialClass {}

Osservazioni

- Le classi parziali devono essere definite all'interno dello stesso assembly e spazio dei nomi, come la classe che stanno estendendo.
- Tutte le parti della classe devono utilizzare la parola chiave `partial` .
- Tutte le parti della classe devono avere la stessa accessibilità; `public` / `protected` / `private` ecc.
- Se una parte utilizza la parola chiave `abstract` , il tipo combinato viene considerato astratto.
- Se una parte utilizza la parola chiave `sealed` , il tipo combinato viene considerato sealed.
- Se una parte usa il tipo base, il tipo combinato eredita da quel tipo.
- Il tipo combinato eredita tutte le interfacce definite su tutte le classi parziali.

Examples

Lezioni parziali

Le classi parziali forniscono la possibilità di dividere la dichiarazione della classe (di solito in file separati). Un problema comune che può essere risolto con classi parziali è che consente agli utenti di modificare il codice generato automaticamente senza temere che le loro modifiche vengano sovrascritte se il codice viene rigenerato. Inoltre, più sviluppatori possono lavorare sulla stessa classe o metodi.

```
using System;

namespace PartialClassAndMethods
{
```

```
    public partial class PartialClass
    {
        public void ExampleMethod() {
            Console.WriteLine("Method call from the first declaration.");
        }
    }

    public partial class PartialClass
    {
        public void AnotherExampleMethod()
        {
            Console.WriteLine("Method call from the second declaration.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.ExampleMethod(); // outputs "Method call from the first declaration."
            partial.AnotherExampleMethod(); // outputs "Method call from the second
declaration."
        }
    }
}
```

Metodi parziali

Il metodo parziale consiste nella definizione in una dichiarazione di classe parziale (come scenario comune - in quello generato automaticamente) e nell'implementazione in un'altra dichiarazione di classe parziale.

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass // Auto-generated
    {
        partial void PartialMethod();
    }

    public partial class PartialClass // Human-written
    {
        public void PartialMethod()
        {
            Console.WriteLine("Partial method called.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.PartialMethod(); // outputs "Partial method called."
        }
    }
```

```
}
```

Classi parziali che ereditano da una classe base

Quando si eredita da qualsiasi classe base, solo una classe parziale deve avere la classe base specificata.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass {}
```

È *possibile* specificare la *stessa* classe base in più di una classe parziale. Verrà contrassegnato come ridondante da alcuni strumenti IDE, ma viene compilato correttamente.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass : BaseClass {} // base class here is redundant
```

Non è *possibile* specificare classi di base *diverse* in più classi parziali, si verificherà un errore del compilatore.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {} // compiler error

// PartialClass2.cs
public partial class PartialClass : OtherBaseClass {} // compiler error
```

Leggi Classe e metodi parziali online: https://riptutorial.com/it/csharp/topic/3674/classe-e-metodi-parziali

Capitolo 25: Classi statiche

Examples

Parola chiave statica

La parola chiave static indica 2 cose:

1. Questo valore non cambia da oggetto a oggetto, ma piuttosto cambia su una classe nel suo complesso
2. Le proprietà e i metodi statici non richiedono un'istanza.

```
public class Foo
{
    public Foo{
        Counter++;
        NonStaticCounter++;
    }

    public static int Counter { get; set; }
    public int NonStaticCounter { get; set; }
}

public class Program
{
    static void Main(string[] args)
    {
        //Create an instance
        var foo1 = new Foo();
        Console.WriteLine(foo1.NonStaticCounter); //this will print "1"

        //Notice this next call doesn't access the instance but calls by the class name.
        Console.WriteLine(Foo.Counter); //this will also print "1"

        //Create a second instance
        var foo2 = new Foo();

        Console.WriteLine(foo2.NonStaticCounter); //this will print "1"

        Console.WriteLine(Foo.Counter); //this will now print "2"
        //The static property incremented on both instances and can persist for the whole
class

    }
}
```

Classi statiche

La parola chiave "statica" quando ci si riferisce a una classe ha tre effetti:

1. Non è **possibile** creare un'istanza di una classe statica (questo rimuove anche il costruttore predefinito)

2. Anche tutte le proprietà e i metodi della classe **devono** essere statici.
3. Una classe `static` è una classe `sealed` , il che significa che non può essere ereditata.

```
public static class Foo
{
    //Notice there is no constructor as this cannot be an instance
    public static int Counter { get; set; }
    public static int GetCount()
    {
        return Counter;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Foo.Counter++;
        Console.WriteLine(Foo.GetCount()); //this will print 1

        //var foo1 = new Foo();
        //this line would break the code as the Foo class does not have a constructor
    }
}
```

Durata della classe statica

Una classe `static` è inizializzata pigramente sull'accesso membro e vive per la durata del dominio dell'applicazione.

```
void Main()
{
    Console.WriteLine("Static classes are lazily initialized");
    Console.WriteLine("The static constructor is only invoked when the class is first
accessed");
    Foo.SayHi();

    Console.WriteLine("Reflecting on a type won't trigger its static .ctor");
    var barType = typeof(Bar);

    Console.WriteLine("However, you can manually trigger it with
System.Runtime.CompilerServices.RuntimeHelpers");
    RuntimeHelpers.RunClassConstructor(barType.TypeHandle);
}

// Define other methods and classes here
public static class Foo
{
    static Foo()
    {
        Console.WriteLine("static Foo.ctor");
    }
    public static void SayHi()
    {
        Console.WriteLine("Foo: Hi");
    }
}
```

```
public static class Bar
{
    static Bar()
    {
        Console.WriteLine("static Bar.ctor");
    }
}
```

Leggi Classi statiche online: https://riptutorial.com/it/csharp/topic/1653/classi-statiche

Capitolo 26: CLSCompliantAttribute

Sintassi

1. [Assembly: CLSCompliant (true)]
2. [CLSCompliant (true)]

Parametri

Costruttore	Parametro
CLSCompliantAttribute (booleano)	Inizializza un'istanza della classe CLSCompliantAttribute con un valore booleano che indica se l'elemento di programma indicato è conforme a CLS.

Osservazioni

Common Language Specification (CLS) è un insieme di regole di base a cui deve essere confermata qualsiasi lingua destinata alla CLI (linguaggio che conferma le specifiche della Common Language Infrastructure) al fine di interoperare con altri linguaggi conformi a CLS.

Elenco delle lingue CLI

Si dovrebbe contrassegnare l'assembly come CLSCompliant nella maggior parte dei casi quando si distribuiscono le librerie. Questo attributo ti garantirà che il tuo codice sarà utilizzabile da tutte le lingue compatibili con CLS. Ciò significa che il tuo codice può essere utilizzato da qualsiasi linguaggio che può essere compilato ed eseguito su CLR (Common Language Runtime)

Quando l'assembly è contrassegnato con `CLSCompliantAttribute` , il compilatore verificherà se il codice viola una qualsiasi delle regole CLS e restituisce un **avviso** se necessario.

Examples

Modificatore di accesso a cui si applicano le regole CLS

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Cat
    {
        internal UInt16 _age = 0;
        private UInt16 _daysTillVacination = 0;
```

```
        //Warning CS3003  Type of 'Cat.DaysTillVacination' is not CLS-compliant
        protected UInt16 DaysTillVacination
        {
            get { return _daysTillVacination; }
        }

        //Warning    CS3003    Type of 'Cat.Age' is not CLS-compliant
        public UInt16 Age
        { get { return _age; } }

        //valid behaviour by CLS-compliant rules
        public int IncreaseAge()
        {
            int increasedAge = (int)_age + 1;

            return increasedAge;
        }

    }
}
```

Le regole per la conformità CLS si applicano solo a componenti pubblici / protetti.

Violazione della regola CLS: tipi non firmati / sbyte

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        internal UInt16 _yearOfCreation = 0;

        //Warning CS3008  Identifier '_numberOfDoors' is not CLS-compliant
        //Warning CS3003  Type of 'Car._numberOfDoors' is not CLS-compliant
        public UInt32 _numberOfDoors = 0;

        //Warning    CS3003    Type of 'Car.YearOfCreation' is not CLS-compliant
        public UInt16 YearOfCreation
        {
            get { return _yearOfCreation; }
        }


        //Warning CS3002  Return type of 'Car.CalculateDistance()' is not CLS-compliant
        public UInt64 CalculateDistance()
        {
            return 0;
        }


        //Warning CS3002  Return type of 'Car.TestDummyUnsignedPointerMethod()' is not CLS-
compliant
        public UIntPtr TestDummyUnsignedPointerMethod()
        {
            int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            UIntPtr ptr = (UIntPtr)arr[0];
```

```
            return ptr;
        }

        //Warning CS3003  Type of 'Car.age' is not CLS-compliant
        public sbyte age = 120;


    }
}
```

Violazione della regola CLS: stessa denominazione

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        //Warning    CS3005    Identifier 'Car.CALCULATEAge()' differing only in case is not
CLS-compliant
        public int CalculateAge()
        {
            return 0;
        }

        public int CALCULATEAge()
        {
            return 0;
        }

    }
}
```

Visual Basic non è sensibile al maiuscolo / minuscolo

Violazione della regola CLS: identificatore _

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        //Warning CS3008  Identifier '_age' is not CLS-complian
        public int _age = 0;
    }

}
```

Non puoi iniziare la variabile con _

Violazione della regola CLS: eredita dalla classe non CLSComplaint

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    [CLSCompliant(false)]
    public class Animal
    {
        public int age = 0;
    }

    //Warning    CS3009    'Dog': base type 'Animal' is not CLS-compliant
    public class Dog : Animal
    {
    }

}
```

Leggi CLSCompliantAttribute online:
https://riptutorial.com/it/csharp/topic/7214/clscompliantattribute

Capitolo 27: Codice Contratti e asserzioni

Examples

Le asserzioni per verificare la logica dovrebbero sempre essere vere

Le asserzioni vengono utilizzate per non eseguire il test dei parametri di input, ma per verificare che il flusso del programma sia corretto, ovvero che è possibile formulare determinate ipotesi sul proprio codice in un determinato momento. In altre parole: un test eseguito con `Debug.Assert` presuppone *sempre* che il valore testato sia `true` .

Debug.Assert esegue solo in build DEBUG; viene filtrato dalle versioni RELEASE. Deve essere considerato uno strumento di debug oltre al test delle unità e non come una sostituzione dei contratti di codice o dei metodi di convalida dell'input.

Ad esempio, questa è una buona asserzione:

```
var systemData = RetrieveSystemConfiguration();
Debug.Assert(systemData != null);
```

Qui affermare è una buona scelta perché possiamo presumere che RetrieveSystemConfiguration () restituirà un valore valido e non restituirà mai null.

Ecco un altro buon esempio:

```
UserData user = RetrieveUserData();
Debug.Assert(user != null);
Debug.Assert(user.Age > 0);
int year = DateTime.Today.Year - user.Age;
```

Innanzitutto, si può presumere che RetrieveUserData () restituisca un valore valido. Quindi, prima di utilizzare la proprietà Age, verifichiamo l'ipotesi (che dovrebbe sempre essere vera) che l'età dell'utente sia strettamente positiva.

Questo è un cattivo esempio di asserzione:

```
string input = Console.ReadLine();
int age = Convert.ToInt32(input);
Debug.Assert(age > 16);
Console.WriteLine("Great, you are over 16");
```

Assert non è per la convalida dell'input perché non è corretto presumere che questa asserzione sarà sempre vera. È necessario utilizzare metodi di convalida dell'input per quello. Nel caso precedente, dovresti anche verificare che il valore di input sia un numero in primo luogo.

Leggi Codice Contratti e asserzioni online: https://riptutorial.com/it/csharp/topic/4349/codice-contratti-e-asserzioni

Capitolo 28: Codice non sicuro in .NET

Osservazioni

- Per poter utilizzare la parola chiave `unsafe` in un progetto .Net, è necessario selezionare "Consenti codice non sicuro" in Proprietà progetto => Build
- L'utilizzo di codice non sicuro può migliorare le prestazioni, tuttavia, è a scapito della sicurezza del codice (quindi il termine `unsafe`).

Ad esempio, quando usi un ciclo for un array in questo modo:

```
for (int i = 0; i < array.Length; i++)
{
    array[i] = 0;
}
```

.NET Framework garantisce che non si superi i limiti dell'array, lanciando una `IndexOutOfRangeException` se l'indice supera i limiti.

Tuttavia, se si utilizza un codice non sicuro, è possibile superare i limiti dell'array in questo modo:

```
unsafe
{
    fixed (int* ptr = array)
    {
        for (int i = 0; i <= array.Length; i++)
        {
            *(ptr+i) = 0;
        }
    }
}
```

Examples

Indice di matrice non sicuro

```
void Main()
{
    unsafe
    {
        int[] a = {1, 2, 3};
        fixed(int* b = a)
        {
            Console.WriteLine(b[4]);
        }
    }
}
```

L'esecuzione di questo codice crea una matrice di lunghezza 3, ma poi tenta di ottenere il quinto elemento (indice 4). Sulla mia macchina, questo stampato `1910457872` , ma il comportamento non è

definito.

Senza il blocco `unsafe` , non è possibile utilizzare i puntatori e, pertanto, non è possibile accedere ai valori oltre la fine di un array senza che venga generata un'eccezione.

Utilizzo non sicuro con gli array

Quando si accede agli array con puntatori, non vi è alcun controllo sui limiti e pertanto non verrà generata `IndexOutOfRangeException` . Questo rende il codice più veloce.

Assegnazione di valori a un array con un puntatore:

```
class Program
{
    static void Main(string[] args)
    {
        unsafe
        {
            int[] array = new int[1000];
            fixed (int* ptr = array)
            {
                for (int i = 0; i < array.Length; i++)
                {
                    *(ptr+i) = i; //assigning the value with the pointer
                }
            }
        }
    }
}
```

Mentre la controparte sicura e normale sarebbe:

```
class Program
{
    static void Main(string[] args)
    {
        int[] array = new int[1000];

        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
        }
    }
}
```

La parte non sicura sarà generalmente più veloce e la differenza di prestazioni può variare in base alla complessità degli elementi nell'array e alla logica applicata a ciascuno. Anche se potrebbe essere più veloce, dovrebbe essere usato con cautela poiché è più difficile da mantenere e più facile da rompere.

Usando non sicuro con le stringhe

```
var s = "Hello";      // The string referenced by variable 's' is normally immutable, but
                      // since it is memory, we could change it if we can access it in an
```

```
                      // unsafe way.

unsafe                // allows writing to memory; methods on System.String don't allow this
{
  fixed (char* c = s) // get pointer to string originally stored in read only memory
    for (int i = 0; i < s.Length; i++)
      c[i] = 'a';     // change data in memory allocated for original string "Hello"
}
Console.WriteLine(s); // The variable 's' still refers to the same System.String
                      // value in memory, but the contents at that location were
                      // changed by the unsafe write above.
                      // Displays: "aaaaa"
```

Leggi Codice non sicuro in .NET online: https://riptutorial.com/it/csharp/topic/81/codice-non-sicuro-in--net

Capitolo 29: Come utilizzare C # Structs per creare un tipo Union (simile a C Unions)

Osservazioni

I tipi dell'Unione sono usati in diverse lingue, in particolare il linguaggio C, per contenere diversi tipi che possono "sovrapporsi" nello stesso spazio di memoria. In altre parole, potrebbero contenere diversi campi che iniziano tutti con lo stesso offset di memoria, anche quando potrebbero avere lunghezze e tipi diversi. Ciò ha il vantaggio sia di risparmiare memoria, sia di eseguire la conversione automatica.

Si prega di notare i commenti nel costruttore della Struct. L'ordine in cui i campi sono inizializzati è estremamente importante. Si desidera innanzitutto inizializzare tutti gli altri campi e quindi impostare il valore che si intende modificare come ultima istruzione. Poiché i campi si sovrappongono, l'ultima impostazione del valore è quella che conta.

Examples

Unioni stile C in C

I tipi di unione sono usati in diverse lingue, come il linguaggio C, per contenere diversi tipi che possono "sovrapporsi". In altre parole, potrebbero contenere diversi campi che iniziano tutti con lo stesso offset di memoria, anche quando potrebbero avere lunghezze e tipi diversi. Ciò ha il vantaggio sia di risparmiare memoria, sia di eseguire la conversione automatica. Pensa ad un indirizzo IP, ad esempio. Internamente, un indirizzo IP viene rappresentato come un numero intero, ma a volte si desidera accedere al diverso componente Byte, come in Byte1.Byte2.Byte3.Byte4. Questo funziona per qualsiasi tipo di valore, sia esso primitivo come Int32 o lungo, o per altre strutture definite dall'utente.

Possiamo ottenere lo stesso effetto in C # usando le strutture di layout esplicito.

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // The "FieldOffset" means that this Integer starts, an offset in bytes.
    // sizeof(int) 4, sizeof(byte) = 1
    [FieldOffset(0)] public int Address;
    [FieldOffset(0)] public byte Byte1;
    [FieldOffset(1)] public byte Byte2;
    [FieldOffset(2)] public byte Byte3;
    [FieldOffset(3)] public byte Byte4;

    public IpAddress(int address) : this()
```

```
    {
        // When we init the Int, the Bytes will change too.
        Address = address;
    }

    // Now we can use the explicit layout to access the
    // bytes separately, without doing any conversion.
    public override string ToString() => $"{Byte1}.{Byte2}.{Byte3}.{Byte4}";
}
```

Avendo definito Struct in questo modo, possiamo usarlo come useremmo un'unione in C. Ad esempio, creiamo un indirizzo IP come numero intero casuale e quindi modifichiamo il primo token dell'indirizzo su "100", cambiandolo da 'ABCD' a '100.BCD':

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"{ip} = {ip.Address}");
ip.Byte1 = 100;
Console.WriteLine($"{ip} = {ip.Address}");
```

Produzione:

```
75.49.5.32 = 537211211
100.49.5.32 = 537211236
```

Visualizza la demo

I tipi di unione in C # possono anche contenere campi Struct

Oltre alle primitive, le strutture di Layout esplicito (Unions) in C #, possono contenere anche altre strutture. Finché un campo è un tipo di valore e non un riferimento, può essere contenuto in un'unione:

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // Same definition of IpAddress, from the example above
}

// Now let's see if we can fit a whole URL into a long

// Let's define a short enum to hold protocols
enum Protocol : short { Http, Https, Ftp, Sftp, Tcp }

// The Service struct will hold the Address, the Port and the Protocol
[StructLayout(LayoutKind.Explicit)]
struct Service
{
    [FieldOffset(0)] public IpAddress Address;
    [FieldOffset(4)] public ushort Port;
    [FieldOffset(6)] public Protocol AppProtocol;
    [FieldOffset(0)] public long Payload;
```

```
    public Service(IpAddress address, ushort port, Protocol protocol)
    {
        Payload = 0;
        Address = address;
        Port  = port;
        AppProtocol = protocol;
    }

    public Service(long payload)
    {
        Address = new IpAddress(0);
        Port = 80;
        AppProtocol = Protocol.Http;
        Payload = payload;
    }

    public Service Copy() => new Service(Payload);

    public override string ToString() => $"{AppProtocol}//{Address}:{Port}/";
}
```

Ora possiamo verificare che l'intera Service Union si adatta alle dimensioni di un lungo (8 byte).

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"Size: {Marshal.SizeOf(ip)} bytes. Value: {ip.Address} = {ip}.");

var s1 = new Service(ip, 8080, Protocol.Https);
var s2 = new Service(s1.Payload);
s2.Address.Byte1 = 100;
s2.AppProtocol = Protocol.Ftp;

Console.WriteLine($"Size: {Marshal.SizeOf(s1)} bytes. Value: {s1.Address} = {s1}.");
Console.WriteLine($"Size: {Marshal.SizeOf(s2)} bytes. Value: {s2.Address} = {s2}.");
```

Visualizza la demo

Leggi Come utilizzare C # Structs per creare un tipo Union (simile a C Unions) online: https://riptutorial.com/it/csharp/topic/5626/come-utilizzare-c-sharp-structs-per-creare-un-tipo-union--simile-a-c-unions-

Capitolo 30: Commenti e regioni

Examples

Commenti

Usare i commenti nei tuoi progetti è un modo pratico per lasciare delle spiegazioni sulle tue scelte progettuali e dovrebbe mirare a rendere la tua vita (o di qualcun altro) più semplice quando si mantiene o si aggiunge al codice.

Ci sono due modi per aggiungere un commento al tuo codice.

Commenti a riga singola

Qualsiasi testo inserito dopo `//` sarà trattato come un commento.

```
public class Program
{
    // This is the entry point of my program.
    public static void Main()
    {
        // Prints a message to the console. - This is a comment!
        System.Console.WriteLine("Hello, World!");

        // System.Console.WriteLine("Hello, World again!"); // You can even comment out code.
        System.Console.ReadLine();
    }
}
```

Commenti multi linea o delimitati

Qualsiasi testo tra `/*` e `*/` sarà trattato come un commento.

```
public class Program
{
    public static void Main()
    {
        /*
            This is a multi line comment
            it will be ignored by the compiler.
        */
        System.Console.WriteLine("Hello, World!");

        // It's also possible to make an inline comment with /* */
        // although it's rarely used in practice
        System.Console.WriteLine(/* Inline comment */ "Hello, World!");

        System.Console.ReadLine();
    }
```

```
}
```

Regioni

Una regione è un blocco di codice comprimibile, che può aiutare con la leggibilità e l'organizzazione del codice.

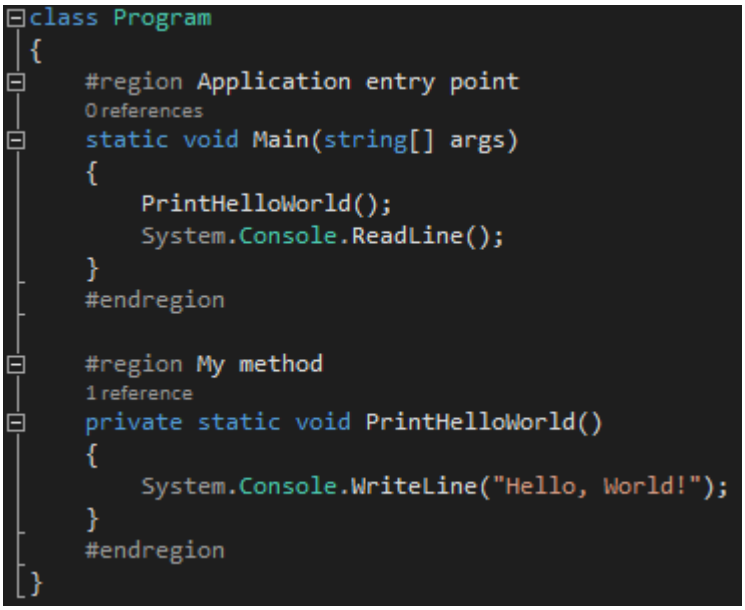
NOTA: la regola StyleCop SA1124 DoNotUseRegions scoraggia l'uso delle regioni. Di solito sono un segno di codice mal organizzato, dato che C # include classi parziali e altre caratteristiche che rendono le regioni obsolete.

Puoi utilizzare le regioni nel seguente modo:

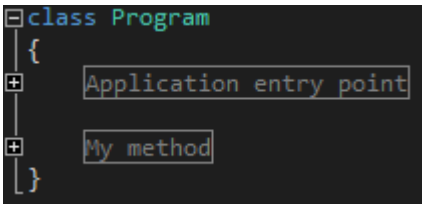
```
class Program
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region My method
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}
```

Quando il codice sopra è visualizzato in un IDE, sarai in grado di comprimere ed espandere il codice usando i simboli + e -.

allargato



crollato

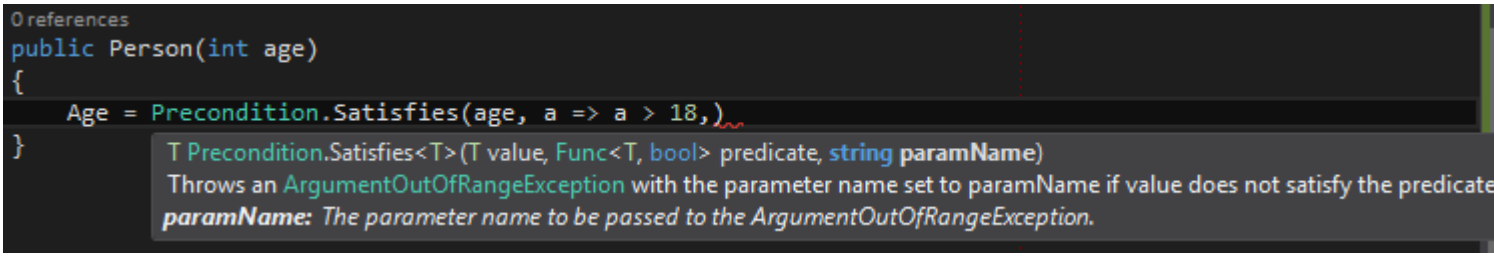


Commenti sulla documentazione

I commenti della documentazione XML possono essere utilizzati per fornire documentazione API che può essere facilmente elaborata dagli strumenti:

```
/// <summary>
/// A helper class for validating method arguments.
/// </summary>
public static class Precondition
{
    /// <summary>
    ///     Throws an <see cref="ArgumentOutOfRangeException"/> with the parameter
    ///     name set to <c>paramName</c> if <c>value</c> does not satisfy the
    ///     <c>predicate</c> specified.
    /// </summary>
    /// <typeparam name="T">
    ///     The type of the argument checked
    /// </typeparam>
    /// <param name="value">
    ///     The argument to be checked
    /// </param>
    /// <param name="predicate">
    ///     The predicate the value is required to satisfy
    /// </param>
    /// <param name="paramName">
    ///     The parameter name to be passed to the
    ///     <see cref="ArgumentOutOfRangeException"/>.
    /// </param>
    /// <returns>The value specified</returns>
    public static T Satisfies<T>(T value, Func<T, bool> predicate, string paramName)
    {
        if (!predicate(value))
            throw new ArgumentOutOfRangeException(paramName);

        return value;
    }
}
```

La documentazione viene immediatamente rilevata da IntelliSense:



Leggi Commenti e regioni online: https://riptutorial.com/it/csharp/topic/5346/commenti-e-regioni

Capitolo 31: Commenti sulla documentazione XML

Osservazioni

Alcune volte è necessario **creare una documentazione di testo estesa** dai tuoi commenti xml. Sfortunatamente ***non esiste un modo standard per farlo*** .

Ma ci sono alcuni progetti separati che puoi usare per questo caso:

- Castello di sabbia
- docu
- NDoc
- DocFX

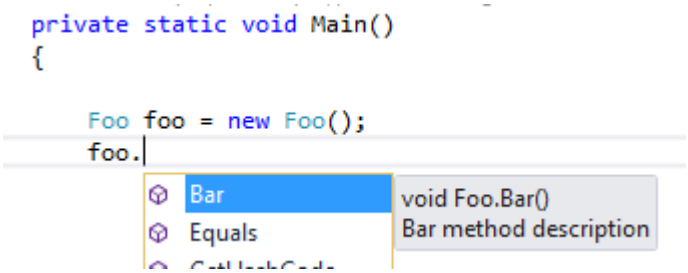
Examples

Annotazione semplice metodo

I commenti della documentazione vengono posizionati direttamente sopra il metodo o la classe che descrivono. Iniziano con tre barre in avanti `///` e consentono la memorizzazione delle meta informazioni tramite XML.

```
/// <summary>
/// Bar method description
/// </summary>
public void Bar()
{

}
```

Le informazioni all'interno dei tag possono essere utilizzate da Visual Studio e altri strumenti per fornire servizi come IntelliSense:



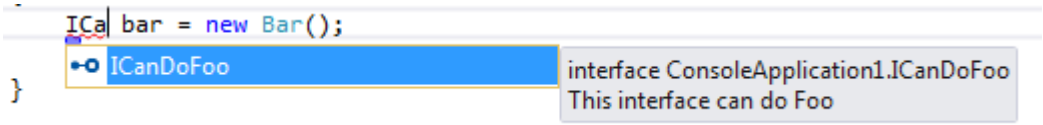
Vedi anche l'elenco di Microsoft di tag di documentazione comuni .

Commenti sulla documentazione di interfaccia e classe

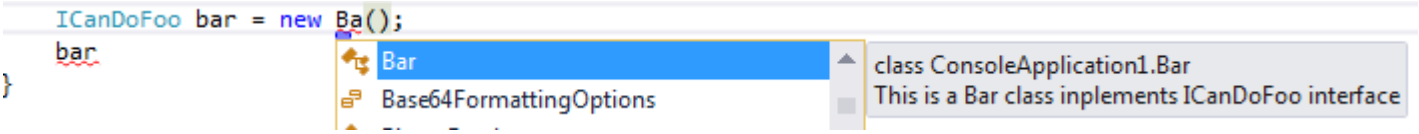
```
/// <summary>
/// This interface can do Foo
/// </summary>
public interface ICanDoFoo
{
    // ...
}

/// <summary>
/// This Bar class implements ICanDoFoo interface
/// </summary>
public class Bar : ICanDoFoo
{
    // ...
}
```

Risultato

Riepilogo dell'interfaccia



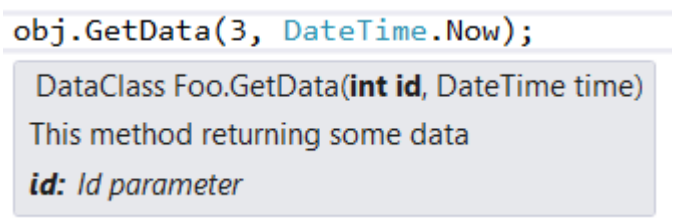
Riepilogo della classe



Commenti sulla documentazione del metodo con elementi param e resi

```
/// <summary>
/// Returns the data for the specified ID and timestamp.
/// </summary>
/// <param name="id">The ID for which to get data. </param>
/// <param name="time">The DateTime for which to get data. </param>
/// <returns>A DataClass instance with the result. </returns>
public DataClass GetData(int id, DateTime time)
{
   // ...
}
```

IntelliSense ti mostra la descrizione per ogni parametro:

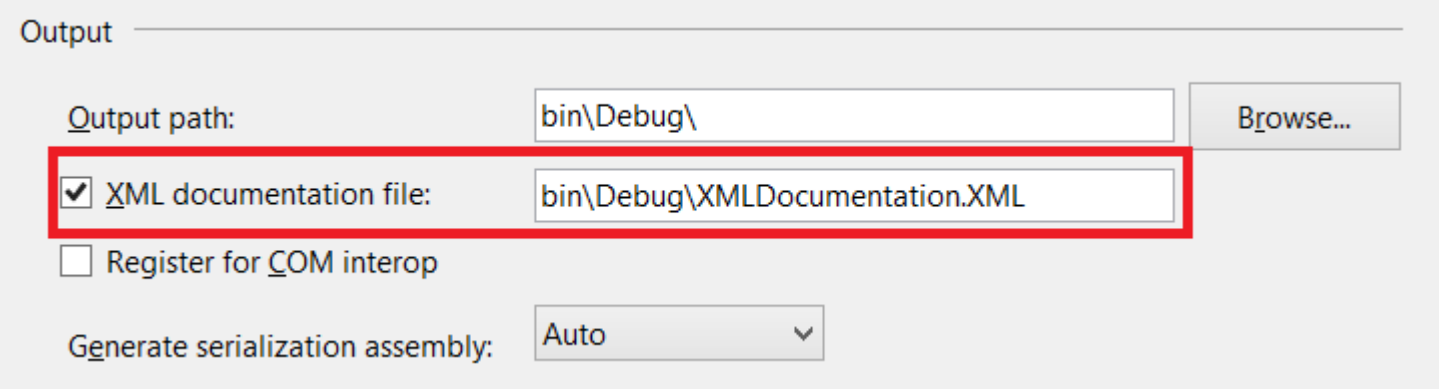


Suggerimento: se Intellisense non viene visualizzato in Visual Studio, eliminare la prima parentesi o la virgola e quindi digitarlo di nuovo.

Generazione di XML dai commenti della documentazione

Per generare un file di documentazione XML dai commenti della documentazione nel codice, utilizzare l'opzione `/doc` con il compilatore `csc.exe` C #.

In Visual Studio 2013/2015, In **Progetto** -> **Proprietà** -> **Crea** -> **Output** , seleziona la casella di controllo del `XML documentation file` :



Quando si genera il progetto, il compilatore produrrà un file XML con un nome corrispondente al nome del progetto (ad es. `XMLDocumentation.dll` -> `XMLDocumentation.xml`).

Quando si utilizza l'assembly in un altro progetto, assicurarsi che il file XML sia nella stessa directory della DLL a cui si fa il riferimento.

Questo esempio:

```
/// <summary>
/// Data class description
/// </summary>
public class DataClass
{
    /// <summary>
    /// Name property description
    /// </summary>
    public string Name { get; set; }
}


/// <summary>
/// Foo function
/// </summary>
public class Foo
{
    /// <summary>
    /// This method returning some data
    /// </summary>
    /// <param name="id">Id parameter</param>
    /// <param name="time">Time parameter</param>
    /// <returns>Data will be returned</returns>
    public DataClass GetData(int id, DateTime time)
    {
        return new DataClass();
    }
}
```

Produce questo xml su build:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>XMLDocumentation</name>
    </assembly>
    <members>
        <member name="T:XMLDocumentation.DataClass">
            <summary>
            Data class description
            </summary>
        </member>
        <member name="P:XMLDocumentation.DataClass.Name">
            <summary>
            Name property description
            </summary>
        </member>
        <member name="T:XMLDocumentation.Foo">
            <summary>
            Foo function
            </summary>
        </member>
        <member name="M:XMLDocumentation.Foo.GetData(System.Int32,System.DateTime)">
            <summary>
            This method returning some data
            </summary>
            <param name="id">Id parameter</param>
            <param name="time">Time parameter</param>
            <returns>Data will be returned</returns>
        </member>
    </members>
</doc>
```

Fare riferimento a un'altra classe nella documentazione

Il tag `<see>` può essere utilizzato per collegarsi a un'altra classe. Contiene il membro `cref` che dovrebbe contenere il nome della classe a cui fare riferimento. Visual Studio fornirà Intellsense durante la scrittura di questo tag e tali riferimenti verranno elaborati anche in caso di ridenominazione della classe di riferimento.

```
/// <summary>
/// You might also want to check out <see cref="SomeOtherClass"/>.
/// </summary>
public class SomeClass
{
}
```

Nei popup di Visual Studio Intellisense tali riferimenti verranno visualizzati anche colorati nel testo.

Per fare riferimento a una classe generica, utilizzare qualcosa di simile al seguente:

```
/// <summary>
/// An enhanced version of <see cref="List{T}"/>.
/// </summary>
public class SomeGenericClass<T>
```

```
{
}
```

Leggi Commenti sulla documentazione XML online:
https://riptutorial.com/it/csharp/topic/740/commenti-sulla-documentazione-xml

Capitolo 32: Compresse le risorse di carattere

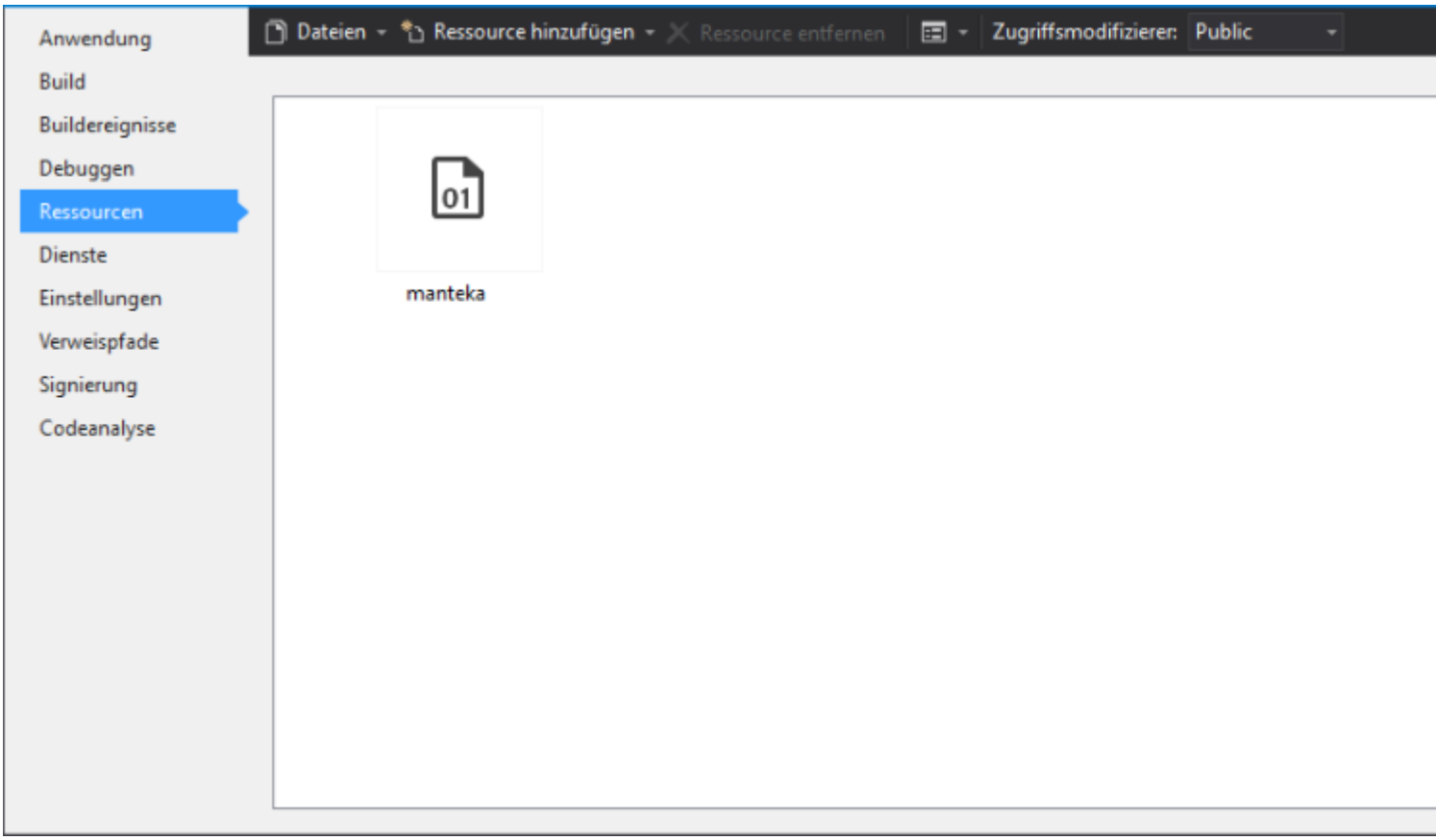
Parametri

Parametro	Dettagli
fontbytes	array di byte dal binario .ttf

Examples

Istanziare 'Fontfamily' da Risorse

```
public FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);
```



Metodo di integrazione

```
public static FontFamily GetResourceFontFamily(byte[] fontbytes)
{
    PrivateFontCollection pfc = new PrivateFontCollection();
    IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
    Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
    pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
    Marshal.FreeCoTaskMem(fontMemPointer);
```

```
    return pfc.Families[0];
}
```

Utilizzo con un "pulsante"

```
    public static class Res
    {
        /// <summary>
        /// URL: https://www.behance.net/gallery/2846011/Manteka
        /// </summary>
        public static FontFamily Maneteke =
GetResourceFontFamily(Properties.Resources.manteka);

        public static FontFamily GetResourceFontFamily(byte[] fontbytes)
        {
            PrivateFontCollection pfc = new PrivateFontCollection();
            IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
            Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
            pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
            Marshal.FreeCoTaskMem(fontMemPointer);
            return pfc.Families[0];
        }
    }

    public class FlatButton : Button
    {
        public FlatButton() : base()
        {
            Font = new Font(Res.Maneteke, Font.Size);
        }

        protected override void OnFontChanged(EventArgs e)
        {
            base.OnFontChanged(e);
            this.Font = new Font(Res.Maneteke, this.Font.Size);
        }
    }
```

Leggi Comprese le risorse di carattere online:
https://riptutorial.com/it/csharp/topic/9789/comprese-le-risorse-di-carattere

Capitolo 33: Contesto di sincronizzazione in attesa asincrona

Examples

Pseudocodice per parole chiave asincrone / attese

Considera un semplice metodo asincrono:

```
async Task Foo()
{
    Bar();
    await Baz();
    Qux();
}
```

Semplificando, possiamo dire che questo codice in realtà significa quanto segue:

```
Task Foo()
{
    Bar();
    Task t = Baz();
    var context = SynchronizationContext.Current;
    t.ContinueWith(task) =>
    {
        if (context == null)
            Qux();
        else
            context.Post((obj) => Qux(), null);
    }, TaskScheduler.Current);

    return t;
}
```

Significa che le parole chiave `async` / `await` il contesto di sincronizzazione corrente se esiste. Ad esempio, è possibile scrivere codice di libreria che funzioni correttamente in applicazioni UI, Web e Console.

Articolo di origine

Disabilitare il contesto di sincronizzazione

Per disabilitare il contesto di sincronizzazione, devi chiamare il metodo `ConfigureAwait` :

```
async Task() Foo()
{
    await Task.Run(() => Console.WriteLine("Test"));
}

. . .
```

```
Foo().ConfigureAwait(false);
```

> ConfigureAwait fornisce un mezzo per evitare il comportamento di cattura predefinito di SynchronizationContext; il passaggio di false per il parametro flowContext impedisce l'utilizzo di SynchronizationContext per riprendere l'esecuzione dopo l'attesa.

Citazione tratta da It's All About the SynchronizationContext .

Perché SynchronizationContext è così importante?

Considera questo esempio:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = RunTooLong();
}
```

Questo metodo bloccherà l'applicazione dell'interfaccia utente fino al completamento di `RunTooLong` . L'applicazione non risponderà.

Puoi provare a eseguire il codice interno in modo asincrono:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() => label1.Text = RunTooLong());
}
```

Ma questo codice non verrà eseguito perché il corpo interno può essere eseguito su thread non UI e non dovrebbe modificare direttamente le proprietà dell'interfaccia utente :

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();

        if (label1.InvokeRequired)
            lable1.BeginInvoke((Action) delegate() { label1.Text = label1Text; });
        else
            label1.Text = label1Text;
    });
}
```

Ora non dimenticare di usare sempre questo modello. Oppure prova `SynchronizationContext.Post` che lo farà per te:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();
```

```
        SynchronizationContext.Current.Post((obj) =>
        {
            label1.Text = label1    Text);
        }, null);
    });
}
```

Leggi Contesto di sincronizzazione in attesa asincrona online:
https://riptutorial.com/it/csharp/topic/7381/contesto-di-sincronizzazione-in-attesa-asincrona

Capitolo 34: Contratti di codice

Sintassi

1. Contract.Requires (Condizione, UserMessage)

   Contract.Requires (Condizione, UserMessage)

   Contract.Result <T>

   Contract.Ensures ()

   Contract.Invariants ()

Osservazioni

.NET supporta l'idea Design by Contract tramite la classe Contracts presente nello spazio dei nomi System.Diagnostics e introdotta in .NET 4.0. Code Contracts API include classi per il controllo statico e di runtime del codice e consente di definire precondizioni, postcondizioni e invarianti all'interno di un metodo. Le condizioni preliminari specificano le condizioni che i parametri devono soddisfare prima che un metodo possa essere eseguito, le postcondizioni verificate al completamento di un metodo e le invarianti definiscono le condizioni che non cambiano durante l'esecuzione di un metodo.

Perché sono necessari i Contratti di codice?

Tenere traccia dei problemi di un'applicazione quando la tua applicazione è in esecuzione, è una delle preoccupazioni principali di tutti gli sviluppatori e amministratori. Il monitoraggio può essere eseguito in molti modi. Per esempio -

- È possibile applicare la traccia sulla nostra applicazione e ottenere i dettagli di un'applicazione quando l'applicazione è in esecuzione
- È possibile utilizzare il meccanismo di registrazione degli eventi quando si esegue l'applicazione. I messaggi possono essere visualizzati utilizzando il Visualizzatore eventi
- È possibile applicare il monitoraggio delle prestazioni dopo un intervallo di tempo specifico e scrivere dati in tempo reale dall'applicazione.

I Contratti di codice utilizzano un approccio diverso per il rilevamento e la gestione dei problemi all'interno di un'applicazione. Invece di convalidare tutto ciò che viene restituito da una chiamata di metodo, Contratti di codice con l'aiuto di precondizioni, postcondizioni e invarianti sui metodi, assicurarsi che tutto ciò che entra e che lasci i metodi sia corretto.

Examples

presupposti

```
namespace CodeContractsDemo
{
    using System;
    using System.Collections.Generic;
    using System.Diagnostics.Contracts;

    public class PaymentProcessor
    {
        private List<Payment> _payments = new List<Payment>();

        public void Add(Payment payment)
        {
            Contract.Requires(payment != null);
            Contract.Requires(!string.IsNullOrEmpty(payment.Name));
            Contract.Requires(payment.Date <= DateTime.Now);
            Contract.Requires(payment.Amount > 0);

            this._payments.Add(payment);
        }
    }
}
```

postconditions

```
public double GetPaymentsTotal(string name)
{
    Contract.Ensures(Contract.Result<double>() >= 0);

    double total = 0.0;

    foreach (var payment in this._payments) {
        if (string.Equals(payment.Name, name)) {
            total += payment.Amount;
        }
    }

    return total;
}
```

invarianti

```
namespace CodeContractsDemo
{
    using System;
    using System.Diagnostics.Contracts;

    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        public Point()
        {
        }
```

```
        public Point(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Set(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Test(int x, int y)
        {
            for (int dx = -x; dx <= x; dx++) {
                this.X = dx;
                Console.WriteLine("Current X = {0}", this.X);
            }

            for (int dy = -y; dy <= y; dy++) {
                this.Y = dy;
                Console.WriteLine("Current Y = {0}", this.Y);
            }

            Console.WriteLine("X = {0}", this.X);
            Console.WriteLine("Y = {0}", this.Y);
        }

        [ContractInvariantMethod]
        private void ValidateCoordinates()
        {
            Contract.Invariant(this.X >= 0);
            Contract.Invariant(this.Y >= 0);
        }
    }
}
```

Definizione dei contratti sull'interfaccia

```
[ContractClass(typeof(ValidationContract))]
interface IValidation
{
    string CustomerID{get;set;}
    string Password{get;set;}
}

[ContractClassFor(typeof(IValidation))]
sealed class ValidationContract:IValidation
{
    string IValidation.CustomerID
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
```

```
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Customer
ID cannot be null!!");
        }
    }

    string IValidation.Password
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Password
cannot be null!!");
        }
    }
}

class Validation:IValidation
{
    public string GetCustomerPassword(string customerID)
    {
        Contract.Requires(!string.IsNullOrEmpty(customerID),"Customer ID cannot be Null");
        Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(customerID),
"Exception!!");
        Contract.Ensures(Contract.Result<string>() != null);
        string password="AAA@1234";
        if (customerID!=null)
        {
            return password;
        }
        else
        {
            return null;
        }

    }

    private string m_custID, m_PWD;

    public string CustomerID
    {
        get
        {
            return m_custID;
        }
        set
        {
            m_custID = value;
        }
    }

    public string Password
    {
        get
        {
            return m_PWD;
        }
        set
```

```
        {
            m_PWD = value;
        }
    }
}
```

Nel codice precedente, abbiamo definito un'interfaccia chiamata `IValidation` con un attributo `[ContractClass]` . Questo attributo prende un indirizzo di una classe in cui abbiamo implementato un contratto per un'interfaccia. La classe `ValidationContract` utilizza le proprietà definite nell'interfaccia e verifica i valori null utilizzando `Contract.Requires<T>` . `T` è una classe di eccezione.

Abbiamo anche contrassegnato l'accesso get con un attributo `[Pure]` . L'attributo puro garantisce che il metodo o una proprietà non modifichi lo stato dell'istanza di una classe in cui è implementata l'interfaccia di `IValidation` .

Leggi Contratti di codice online: https://riptutorial.com/it/csharp/topic/4241/contratti-di-codice

Capitolo 35: Convenzioni di denominazione

introduzione

Questo argomento delinea alcune convenzioni di denominazione di base utilizzate durante la scrittura nel linguaggio C #. Come tutte le convenzioni, non sono applicate dal compilatore, ma assicurano la leggibilità tra gli sviluppatori.

Per le linee guida complete sulla progettazione di framework .NET, consultare docs.microsoft.com/dotnet/standard/design-guidelines .

Osservazioni

Scegli nomi di identificatori facilmente leggibili

Ad esempio, una proprietà denominata HorizontalAlignment è più leggibile in inglese di AlignmentHorizontal.

Favorire la leggibilità per brevità

Il nome della proprietà `CanScrollHorizontally` è migliore di `ScrollableX` (un riferimento oscuro all'asse X).

Evitare l'uso di caratteri di sottolineatura, trattini o altri caratteri non alfanumerici.

Non usare la notazione ungherese

La notazione ungherese è la pratica di includere un prefisso negli identificatori per codificare alcuni metadati relativi al parametro, come il tipo di dati dell'identificatore, ad esempio `string strName` .

Inoltre, evita l'uso di identificatori in conflitto con le parole chiave già utilizzate in C #.

Abbreviazioni e Acronimi

In generale, non si dovrebbero usare abbreviazioni o acronimi; questi rendono i tuoi nomi meno leggibili. Allo stesso modo, è difficile sapere quando è sicuro assumere che un acronimo è ampiamente riconosciuto.

Examples

Convenzioni sulla capitalizzazione

I seguenti termini descrivono diversi modi per identificare i casi.

Involucro Pascal

La prima lettera nell'identificatore e la prima lettera di ogni parola concatenata successiva sono in maiuscolo. È possibile utilizzare il caso Pascal per identificatori di tre o più caratteri. Ad esempio: `BackColor`

Camel Casing

La prima lettera di un identificatore è in minuscolo e la prima lettera di ogni parola concatenata successiva è in maiuscolo. Ad esempio: `backColor`

Lettere maiuscole

Tutte le lettere nell'identificatore sono in maiuscolo. Ad esempio: `IO`

Regole

Quando un identificatore è costituito da più parole, non utilizzare separatori, come caratteri di sottolineatura ("_") o trattini ("-"), tra le parole. Invece, usa l'involucro per indicare l'inizio di ogni parola.

La tabella seguente riepiloga le regole di maiuscole per gli identificatori e fornisce esempi per i diversi tipi di identificatori:

Identifier	Astuccio	Esempio
Variabile locale	Cammello	carName
Classe	Pascal	AppDomain
Tipo di enumerazione	Pascal	ErrorLevel
Valori di enumerazione	Pascal	Errore fatale
Evento	Pascal	ValueChanged
Classe di eccezione	Pascal	WebException
Campo statico di sola lettura	Pascal	RedValue
Interfaccia	Pascal	IDisposable
Metodo	Pascal	Accordare
Spazio dei nomi	Pascal	System.Drawing

Identifier	Astuccio	Esempio
Parametro	Cammello	typeName
Proprietà	Pascal	Colore di sfondo

Ulteriori informazioni possono essere trovate su MSDN .

interfacce

Le interfacce devono essere denominate con nomi o frasi di nomi o aggettivi che descrivono il comportamento. Per esempio `IComponent` usa un nome descrittivo, `ICustomAttributeProvider` usa una frase nominale e `IPersistable` usa un aggettivo.

I nomi dell'interfaccia devono essere preceduti dalla lettera `I` , per indicare che il tipo è un'interfaccia e deve essere utilizzato il caso Pascal.

Di seguito sono indicate le interfacce correttamente denominate:

```
public interface IServiceProvider
public interface IFormatable
```

Campi privati

Esistono due convenzioni comuni per i campi privati: `camelCase` e `_camelCaseWithLeadingUnderscore` .

Cassa del cammello

```
public class Rational
{
    private readonly int numerator;
    private readonly int denominator;

    public Rational(int numerator, int denominator)
    {
        // "this" keyword is required to refer to the class-scope field
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

Cassa del cammello con underscore

```
public class Rational
{
    private readonly int _numerator;
    private readonly int _denominator;

    public Rational(int numerator, int denominator)
    {
```

```
        // Names are unique, so "this" keyword is not required
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

Namespace

Il formato generale per gli spazi dei nomi è:

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>].
```

Esempi inclusi:

```
Fabrikam.Math
Litware.Security
```

Il prefisso dei nomi di spazi dei nomi con il nome di una società impedisce agli spazi dei nomi di società diverse di avere lo stesso nome.

Enums

Usa un nome singolare per la maggior parte degli Enum

```
public enum Volume
{
   Low,
   Medium,
   High
}
```

Utilizzare un nome plurale per i tipi Enum che sono campi bit

```
[Flags]
public enum MyColors
{
    Yellow = 1,
    Green = 2,
    Red = 4,
    Blue = 8
}
```

Nota: aggiungi sempre `FlagsAttribute` *a un campo Enum di bit.*

Non aggiungere 'enum' come suffisso

```
public enum VolumeEnum // Incorrect
```

Non usare il nome enum in ciascuna voce

```
public enum Color
{
    ColorBlue, // Remove Color, unnecessary
    ColorGreen,
}
```

eccezioni

Aggiungi 'eccezione' come suffisso

I nomi delle eccezioni personalizzate dovrebbero essere suffissi con "-Exception".

Di seguito sono riportate correttamente le eccezioni:

```
public class MyCustomException : Exception
public class FooException : Exception
```

Leggi Convenzioni di denominazione online:
https://riptutorial.com/it/csharp/topic/2330/convenzioni-di-denominazione

Capitolo 36: Costrutti di flusso di dati di Task Parallel Library (TPL)

Examples

JoinBlock

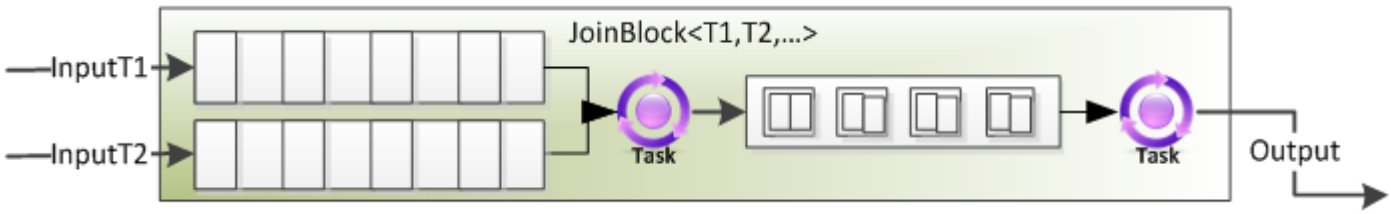
(Raccoglie 2-3 input e li combina in una tupla)

Come BatchBlock, JoinBlock <T1, T2, ...> è in grado di raggruppare i dati da più origini dati. In effetti, questo è lo scopo principale di JoinBlock <T1, T2, ...>.

Ad esempio, un JoinBlock <string, double, int> è un ISourceBlock <Tuple <string, double, int >>.

Come con BatchBlock, JoinBlock <T1, T2, ...> è in grado di funzionare sia in modalità avida che non avida.

- Nella modalità greedy di default, tutti i dati offerti ai target sono accettati, anche se l'altro target non ha i dati necessari con cui formare una tupla.
- In modalità non greedy, i target del blocco posticiperanno i dati fino a quando a tutti i target non saranno stati offerti i dati necessari per creare una tupla, a quel punto il blocco si impegnerà in un protocollo di commit a due fasi per recuperare atomicamente tutti gli elementi necessari dalle fonti. Questo rinvio consente a un'altra entità di consumare i dati nel frattempo in modo da consentire al sistema complessivo di avanzare.



Elaborazione di richieste con un numero limitato di oggetti raggruppati

```
var throttle = new JoinBlock<ExpensiveObject, Request>();
for(int i=0; i<10; i++)
{
    requestProcessor.Target1.Post(new ExpensiveObject());
}

var processor = new Transform<Tuple<ExpensiveObject, Request>, ExpensiveObject>(pair =>
{
    var resource = pair.Item1;
    var request = pair.Item2;

    request.ProcessWith(resource);

    return resource;
});
```

```
throttle.LinkTo(processor);
processor.LinkTo(throttle.Target1);
```

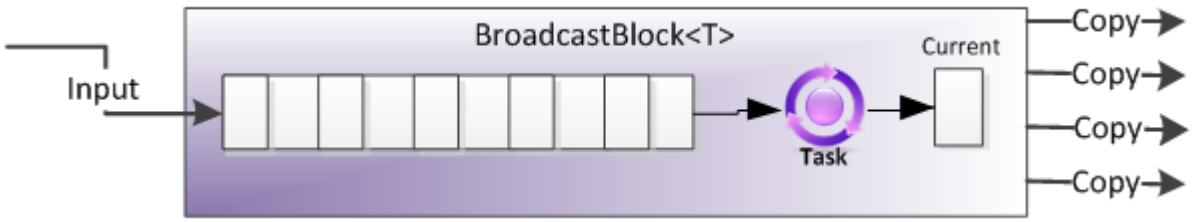
Introduzione a TPL Dataflow di Stephen Toub

BroadcastBlock

(Copia un elemento e invia le copie a ogni blocco a cui è collegato)

A differenza di BufferBlock, la missione di BroadcastBlock nella vita è di abilitare tutti i target collegati dal blocco per ottenere una copia di ogni elemento pubblicato, sovrascrivendo continuamente il valore "corrente" con quelli propagati ad esso.

Inoltre, a differenza di BufferBlock, BroadcastBlock non mantiene i dati inutilmente. Dopo che un particolare dato è stato offerto a tutti i bersagli, quell'elemento sarà sovrascritto da qualunque pezzo di dati è in linea (come con tutti i blocchi di flusso di dati, i messaggi sono gestiti in ordine FIFO). Questo elemento sarà offerto a tutti gli obiettivi, e così via.



Producer / consumer asincrono con un produttore Throttled

```
var ui = TaskScheduler.FromCurrentSynchronizationContext();
var bb = new BroadcastBlock<ImageData>(i => i);

var saveToDiskBlock = new ActionBlock<ImageData>(item =>
    item.Image.Save(item.Path)
);

var showInUiBlock = new ActionBlock<ImageData>(item =>
    imagePanel.AddImage(item.Image),
    new DataflowBlockOptions { TaskScheduler =
TaskScheduler.FromCurrentSynchronizationContext() }
);

bb.LinkTo(saveToDiskBlock);
bb.LinkTo(showInUiBlock);
```

Esposizione dello stato da un agente

```
public class MyAgent
{
    public ISourceBlock<string> Status { get; private set; }

    public MyAgent()
    {
```

```
        Status = new BroadcastBlock<string>();
        Run();
    }

    private void Run()
    {
        Status.Post("Starting");
        Status.Post("Doing cool stuff");
        …
        Status.Post("Done");
    }
}
```

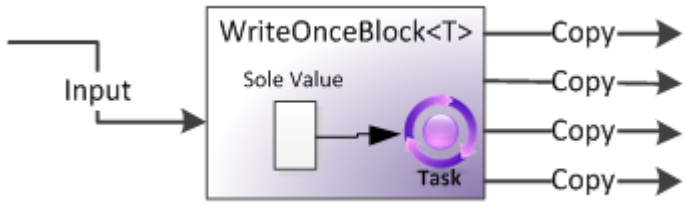
Introduzione a TPL Dataflow di Stephen Toub

WriteOnceBlock

(Variabile di sola lettura: memorizza il suo primo elemento di dati e ne distribuisce copie come output. Ignora tutti gli altri elementi di dati)

Se BufferBlock è il blocco più fondamentale in TPL Dataflow, WriteOnceBlock è il più semplice. Memorizza al massimo un valore, e una volta che questo valore è stato impostato, non verrà mai sostituito o sovrascritto.

Puoi pensare a WriteOnceBlock come simile a una variabile membro readonly in C #, ma invece di essere solo impostabile in un costruttore e quindi immutabile, è impostabile solo una volta ed è quindi immutabile.



Dividere i potenziali output di un'attività

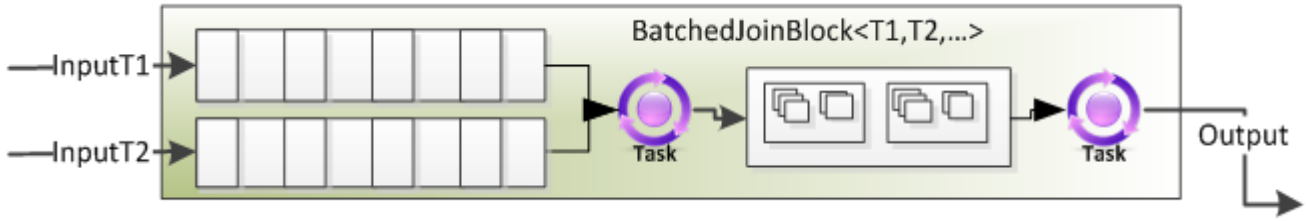
```
public static async void SplitIntoBlocks(this Task<T> task,
    out IPropagatorBlock<T> result,
    out IPropagatorBlock<Exception> exception)
{
    result = new WriteOnceBlock<T>(i => i);
    exception = new WriteOnceBlock<Exception>(i => i);

    try
    {
        result.Post(await task);
    }
    catch(Exception ex)
    {
        exception.Post(ex);
    }
}
```

Introduzione a TPL Dataflow di Stephen Toub

BatchedJoinBlock

(Raccoglie un certo numero di elementi totali da 2-3 input e li raggruppa in una tupla di raccolte di elementi di dati)

BatchedJoinBlock <T1, T2, ...> è in un certo senso una combinazione di BatchBlock e JoinBlock <T1, T2, ...>.
Mentre JoinBlock <T1, T2, ...> viene utilizzato per aggregare un input da ciascun target in una tupla e BatchBlock viene utilizzato per aggregare N input in una raccolta, BatchJoinBlock <T1, T2, ...> viene utilizzato per raccogliere N input da across tutti gli obiettivi in tuple di raccolte.



Scatter / gather

Si consideri un problema di dispersione / raccolta in cui vengono avviate N operazioni, alcune delle quali possono avere successo e produrre output di stringhe, e altre potrebbero non riuscire e produrre eccezioni.

```
var batchedJoin = new BatchedJoinBlock<string, Exception>(10);

for (int i=0; i<10; i++)
{
    Task.Factory.StartNew(() => {
        try { batchedJoin.Target1.Post(DoWork()); }
        catch(Exception ex) { batchJoin.Target2.Post(ex); }
    });
}

var results = await batchedJoin.ReceiveAsync();

foreach(string s in results.Item1)
{
    Console.WriteLine(s);
}

foreach(Exception e in results.Item2)
{
    Console.WriteLine(e);
}
```

Introduzione a TPL Dataflow di Stephen Toub

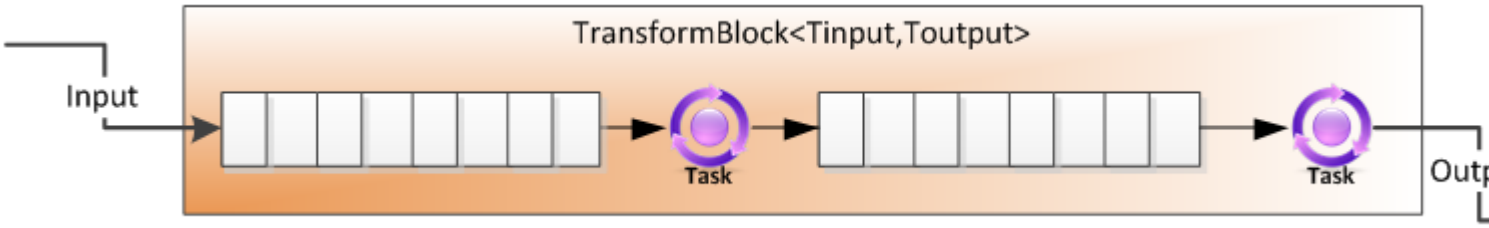
TransformBlock

(Seleziona, uno a uno)

Come con ActionBlock, TransformBlock <TInput, TOutput> consente l'esecuzione di un delegato per eseguire alcune azioni per ogni dato di input; **a differenza di ActionBlock, questa elaborazione ha un output.** Questo delegato può essere un Func <TInput, TOutput>, nel qual caso l'elaborazione di quell'elemento viene considerata completata al ritorno del delegato oppure può essere Func <TInput, Task>, nel qual caso l'elaborazione di quell'elemento viene considerata completata quando il delegato ritorna ma quando l'attività restituita è completa. Per chi ha familiarità con LINQ, è in qualche modo simile a Select () in quanto richiede un input, trasforma quell'input in qualche modo e quindi produce un output.

Per impostazione predefinita, TransformBlock <TInput, TOutput> elabora i suoi dati in modo sequenziale con un MaxDegreeOfParallelism uguale a 1. Oltre a ricevere l'input bufferizzato ed elaborarlo, questo blocco prenderà tutti i suoi output elaborati e anche il buffer (dati che non sono stati elaborati e dati che sono stati elaborati).

Ha 2 compiti: uno per elaborare i dati e uno per spingere i dati al blocco successivo.



Una pipeline concorrente

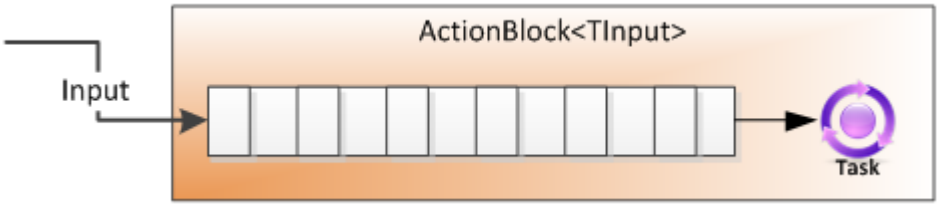
```
var compressor = new TransformBlock<byte[], byte[]>(input => Compress(input));
var encryptor = new TransformBlock<byte[], byte[]>(input => Encrypt(input));

compressor.LinkTo(Encryptor);
```

Introduzione a TPL Dataflow di Stephen Toub

ActionBlock

(per ciascuno)

Questa classe può essere considerata logicamente come un buffer per i dati da elaborare in combinazione con le attività per l'elaborazione di tali dati, con il "blocco del flusso di dati" che gestisce entrambi. Nel suo uso più basilare, possiamo istanziare un oggetto ActionBlock e "postare" i dati su di esso; il delegato fornito alla costruzione di ActionBlock verrà eseguito in modo asincrono per ogni dato inviato.



Calcolo sincrono

```
var ab = new ActionBlock<TInput>(i =>
{
    Compute(i);
});
…
ab.Post(1);
ab.Post(2);
ab.Post(3);
```

Limitazione di download asincroni fino a un massimo di 5 contemporaneamente

```
var downloader = new ActionBlock<string>(async url =>
{
    byte [] imageData = await DownloadAsync(url);
    Process(imageData);
}, new DataflowBlockOptions { MaxDegreeOfParallelism = 5 });

downloader.Post("http://website.com/path/to/images");
downloader.Post("http://another-website.com/path/to/images");
```

Introduzione a TPL Dataflow di Stephen Toub

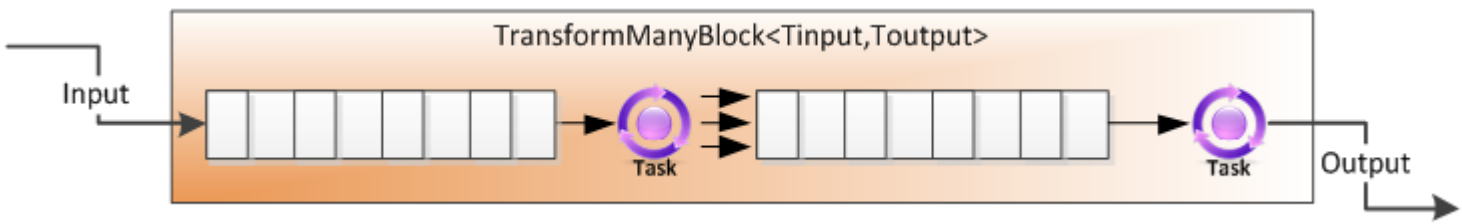
TransformManyBlock

(SelectMany, 1-m: i risultati di questa mappatura sono "appiattiti", proprio come SelectMany di LINQ)

TransformManyBlock <TInput, TOutput> è molto simile a TransformBlock <TInput, TOutput>. La differenza principale è che mentre TransformBlock <TInput, TOutput> produce uno e un solo output per ogni input, TransformManyBlock <TInput, TOutput> produce qualsiasi numero (zero o più) output per ogni input. Come con ActionBlock e TransformBlock <TInput, TOutput>, questa elaborazione può essere specificata utilizzando i delegati, sia per l'elaborazione sincrona che asincrona.

Un Func <TInput, IEnumerable> viene utilizzato per synchronous e un Func <TInput, Task <IEnumerable >> viene utilizzato per asincrono. Come con ActionBlock e TransformBlock <TInput, TOutput>, TransformManyBlock <TInput, TOutput> passa automaticamente all'elaborazione sequenziale, ma può essere configurato diversamente.

Il delegato di mappatura retunisce una raccolta di elementi, che vengono inseriti singolarmente nel buffer di output.



Web crawler asincrono

```
var downloader = new TransformManyBlock<string, string>(async url =>
{
    Console.WriteLine("Downloading " + url);
    try
    {
        return ParseLinks(await DownloadContents(url));
    }
    catch{}

    return Enumerable.Empty<string>();
});
downloader.LinkTo(downloader);
```

Espandere un Enumerable nei suoi elementi costitutivi

```
var expanded = new TransformManyBlock<T[], T>(array => array);
```

Filtraggio passando da 1 a 0 o 1 elemento

```
public IPropagatorBlock<T> CreateFilteredBuffer<T>(Predicate<T> filter)
{
    return new TransformManyBlock<T, T>(item =>
        filter(item) ? new [] { item } : Enumerable.Empty<T>());
}
```

Introduzione a TPL Dataflow di Stephen Toub

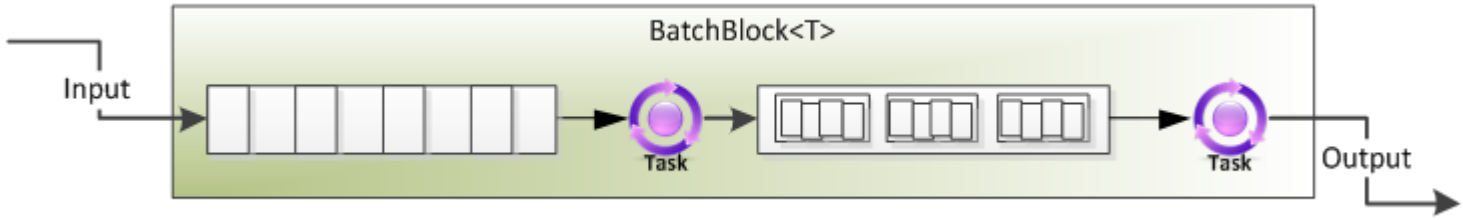
BatchBlock

(Raggruppa un certo numero di elementi di dati sequenziali in raccolte di elementi di dati)

BatchBlock combina N singoli elementi in un unico articolo, rappresentato come una matrice di elementi. Un'istanza viene creata con una specifica dimensione di batch e il blocco crea quindi un batch non appena viene ricevuto quel numero di elementi, in modo asincrono che invia il batch al buffer di output.

BatchBlock è in grado di eseguire in entrambe le modalità avide e non avide.

- Nella modalità greedy di default, tutti i messaggi offerti al blocco da qualsiasi numero di fonti sono accettati e memorizzati nel buffer per essere convertiti in batch.
-
    - In modalità non avida, tutti i messaggi vengono posticipati dalle fonti finché un numero sufficiente di fonti non ha offerto messaggi al blocco per creare un batch. Quindi, un BatchBlock può essere usato per ricevere 1 elemento da ciascuna delle fonti N, da N elementi da 1 fonte e da una miriade di opzioni intermedie.



Richieste di raggruppamento in gruppi di 100 da inviare a un database

```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));

batchRequests.LinkTo(sendToDb);
```

Creare un batch una volta al secondo

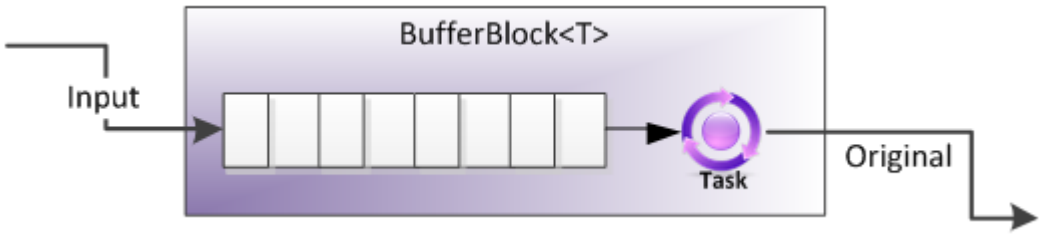
```
var batch = new BatchBlock<T>(batchSize:Int32.MaxValue);
new Timer(() => { batch.TriggerBatch(); }).Change(1000, 1000);
```

Introduzione a TPL Dataflow di Stephen Toub

BufferBlock

(Coda FIFO: i dati che arrivano sono i dati che escono)

In breve, BufferBlock fornisce un buffer illimitato o limitato per la memorizzazione di istanze di T.
È possibile "postare" istanze di T al blocco, il che fa sì che i dati inviati vengano memorizzati in un ordine FIFO (first-in-first-out) dal blocco.
È possibile "ricevere" dal blocco, che consente di ottenere in modo sincrono o asincrono istanze di T precedentemente memorizzate o disponibili in futuro (di nuovo, FIFO).



Producer / consumer asincrono con un produttore Throttled

```
// Hand-off through a bounded BufferBlock<T>
private static BufferBlock<int> _Buffer = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 10 });

// Producer
private static async void Producer()
{
    while(true)
    {
        await _Buffer.SendAsync(Produce());
```

```
    }
}

// Consumer
private static async Task Consumer()
{
    while(true)
    {
        Process(await _Buffer.ReceiveAsync());
    }
}

// Start the Producer and Consumer
private static async Task Run()
{
    await Task.WhenAll(Producer(), Consumer());
}
```

Introduzione a TPL Dataflow di Stephen Toub

Leggi Costrutti di flusso di dati di Task Parallel Library (TPL) online: https://riptutorial.com/it/csharp/topic/3110/costrutti-di-flusso-di-dati-di-task-parallel-library--tpl-

Capitolo 37: Costruttori e Finalizzatori

introduzione

I costruttori sono metodi in una classe che vengono invocati quando viene creata un'istanza di quella classe. La loro principale responsabilità è quella di lasciare il nuovo oggetto in uno stato utile e coerente.

I distruttori / finalizzatori sono metodi di una classe invocati quando un'istanza viene distrutta. In C # raramente vengono scritti / utilizzati esplicitamente.

Osservazioni

C # non ha effettivamente distruttori, ma piuttosto Finalizzatori che usano la sintassi del distruttore di stile C ++. La specifica di un distruttore sovrascrive il metodo `Object.Finalize()` che non può essere chiamato direttamente.

A differenza di altri linguaggi con sintassi simile, questi metodi *non* vengono chiamati quando gli oggetti escono dall'ambito, ma vengono chiamati quando viene eseguito il Garbage Collector, che si verifica in determinate condizioni . Come tali, essi *non* sono garantiti per l'esecuzione in un ordine particolare.

I finalizzatori dovrebbero essere responsabili della pulizia **solo delle** risorse non gestite (puntatori acquisiti tramite la classe Marshal, ricevuti tramite p / Invoke (chiamate di sistema) o puntatori grezzi utilizzati all'interno di blocchi non sicuri). Per ripulire le risorse gestite, rivedere IDisposable, il pattern Dispose e l'istruzione `using` .

(Ulteriori letture: quando dovrei creare un distruttore?)

Examples

Costruttore predefinito

Quando un tipo è definito senza un costruttore:

```
public class Animal
{
}
```

quindi il compilatore genera un costruttore predefinito equivalente al seguente:

```
public class Animal
{
    public Animal() {}
}
```

La definizione di qualsiasi costruttore per il tipo sopprimerà la generazione predefinita del costruttore. Se il tipo è stato definito come segue:

```
public class Animal
{
    public Animal(string name) {}
}
```

quindi un `Animal` può essere creato solo chiamando il costruttore dichiarato.

```
// This is valid
var myAnimal = new Animal("Fluffy");
// This fails to compile
var unnamedAnimal = new Animal();
```

Per il secondo esempio, il compilatore mostrerà un messaggio di errore:

> 'Animale' non contiene un costruttore che accetta 0 argomenti

Se si desidera che una classe abbia sia un costruttore senza parametri sia un costruttore che accetta un parametro, è possibile farlo implementando esplicitamente entrambi i costruttori.

```
public class Animal
{

    public Animal() {} //Equivalent to a default constructor.
    public Animal(string name) {}
}
```

Il compilatore non sarà in grado di generare un costruttore predefinito se la classe estende un'altra classe che non ha un costruttore senza parametri. Ad esempio, se avessimo una `Creature` classe:

```
public class Creature
{
    public Creature(Genus genus) {}
}
```

quindi `Animal` definito come `class Animal : Creature {}` non verrebbe compilato.

Chiamare un costruttore da un altro costruttore

```
public class Animal
{
    public string Name { get; set; }

    public Animal() : this("Dog")
    {
    }

    public Animal(string name)
    {
        Name = name;
    }
```

```
}

var dog = new Animal();      // dog.Name will be set to "Dog" by default.
var cat = new Animal("Cat"); // cat.Name is "Cat", the empty constructor is not called.
```

Costruttore statico

Un costruttore statico viene chiamato la prima volta che viene inizializzato un membro di un tipo, viene chiamato un membro di classe statico o un metodo statico. Il costruttore statico è thread-safe. Un costruttore statico è comunemente usato per:

- Inizializza lo stato statico, ovvero lo stato che è condiviso tra diverse istanze della stessa classe.
- Crea un singleton

Esempio:

```
class Animal
{
    // * A static constructor is executed only once,
    //   when a class is first accessed.
    // * A static constructor cannot have any access modifiers
    // * A static constructor cannot have any parameters
    static Animal()
    {
        Console.WriteLine("Animal initialized");
    }

    // Instance constructor, this is executed every time the class is created
    public Animal()
    {
        Console.WriteLine("Animal created");
    }

    public static void Yawn()
    {
        Console.WriteLine("Yawn!");
    }
}

var turtle = new Animal();
var giraffe = new Animal();
```

Produzione:

Animale inizializzato
Animale creato
Animale creato

Visualizza la demo

Se la prima chiamata riguarda un metodo statico, il costruttore statico viene richiamato senza il costruttore dell'istanza. Questo è OK, perché il metodo statico non può accedere allo stato di istanza in ogni caso.

```
Animal.Yawn();
```

Questo produrrà:

> Animale inizializzato
> Sbadiglio!

Vedi anche Eccezioni in costruttori statici e Generic Static Constructors .

Esempio Singleton:

```
public class SessionManager
{
    public static SessionManager Instance;

    static SessionManager()
    {
        Instance = new SessionManager();
    }
}
```

Chiamando il costruttore della classe base

Un costruttore di una classe base viene chiamato prima che venga eseguito un costruttore di una classe derivata. Ad esempio, se `Mammal` estende `Animal` , il codice contenuto nel costruttore di `Animal` viene chiamato prima quando si crea un'istanza di un `Mammal` .

Se una classe derivata non specifica esplicitamente quale costruttore della classe base deve essere chiamato, il compilatore assume il costruttore senza parametri.

```
public class Animal
{
    public Animal() { Console.WriteLine("An unknown animal gets born."); }
    public Animal(string name) { Console.WriteLine(name + " gets born"); }
}

public class Mammal : Animal
{
    public Mammal(string name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

In questo caso, l'istanzializzazione di un `Mammal` chiamando il `new Mammal("George the Cat")` verrà stampata

> Un animale sconosciuto nasce.
> George the Cat è un mammifero.

Visualizza la demo

La chiamata a un diverso costruttore della classe base viene effettuata posizionando `: base(args)`

tra la firma del costruttore e il suo corpo:

```
public class Mammal : Animal
{
    public Mammal(string name) : base(name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

Chiamando il `new Mammal("George the Cat")` ora stamperà:

> George the Cat nasce.
> George the Cat è un mammifero.

Visualizza la demo

Finalizzatori su classi derivate

Quando un grafico dell'oggetto è finalizzato, l'ordine è il contrario della costruzione. Ad esempio, il super-tipo è finalizzato prima del tipo base come dimostra il seguente codice:

```
class TheBaseClass
{
    ~TheBaseClass()
    {
        Console.WriteLine("Base class finalized!");
    }
}

class TheDerivedClass : TheBaseClass
{
    ~TheDerivedClass()
    {
        Console.WriteLine("Derived class finalized!");
    }
}

//Don't assign to a variable
//to make the object unreachable
new TheDerivedClass();

//Just to make the example work;
//this is otherwise NOT recommended!
GC.Collect();

//Derived class finalized!
//Base class finalized!
```

Modello di costruttore Singleton

```
public class SingletonClass
{
    public static SingletonClass Instance { get; } = new SingletonClass();
```

```
    private SingletonClass()
    {
        // Put custom constructor code here
    }
}
```

Poiché il costruttore è privato, non è possibile creare nuove istanze di `SingletonClass` mediante il consumo di codice. L'unico modo per accedere alla singola istanza di `SingletonClass` consiste nell'utilizzare la proprietà statica `SingletonClass.Instance` .

La proprietà `Instance` è assegnata da un costruttore statico generato dal compilatore C #. Il runtime .NET garantisce che il costruttore statico venga eseguito al massimo una volta e venga eseguito prima della prima lettura `Instance` . Pertanto, tutti i problemi di sincronizzazione e di inizializzazione vengono eseguiti dal runtime.

Nota che se il costruttore statico fallisce, la classe `Singleton` diventa definitivamente inutilizzabile per la vita di AppDomain.

Inoltre, non è garantito il funzionamento del costruttore statico al momento del primo accesso di `Instance` . Piuttosto, funzionerà *prima o poi* . Questo rende il momento in cui l'inizializzazione avviene non deterministica. Nei casi pratici, il JIT chiama spesso il costruttore statico durante la *compilazione* (non l'esecuzione) di un metodo che fa riferimento `Instance` . Questa è un'ottimizzazione delle prestazioni.

Vedere la pagina Implementazioni Singleton per altri modi per implementare il modello singleton.

Forzare un costruttore statico da chiamare

Sebbene i costruttori statici vengano sempre chiamati prima del primo utilizzo di un tipo, talvolta è utile poterli forzare a chiamare e la classe `RuntimeHelpers` fornisce un helper per questo:

```
using System.Runtime.CompilerServices;
// ...
RuntimeHelpers.RunClassConstructor(typeof(Foo).TypeHandle);
```

Osservazione **:** verrà eseguita tutta l'inizializzazione statica (ad esempio gli inizializzatori dei campi), non solo il costruttore stesso.

Usi potenziali **:** forzatura dell'inizializzazione durante la schermata iniziale in un'applicazione di interfaccia utente o per garantire che un costruttore statico non fallisca in un test di unità.

Chiamare metodi virtuali nel costruttore

A differenza di C ++ in C # è possibile chiamare un metodo virtuale dal costruttore di classi (OK, è possibile anche in C ++ ma il comportamento all'inizio è sorprendente). Per esempio:

```
abstract class Base
{
    protected Base()
    {
```

```
        _obj = CreateAnother();
    }

    protected virtual AnotherBase CreateAnother()
    {
        return new AnotherBase();
    }

    private readonly AnotherBase _obj;
}

sealed class Derived : Base
{
    public Derived() { }

    protected override AnotherBase CreateAnother()
    {
        return new AnotherDerived();
    }
}

var test = new Derived();
// test._obj is AnotherDerived
```

Se provieni da uno sfondo C ++ questo è sorprendente, il costruttore della classe base vede già la tabella del metodo virtuale della classe derivata!

Attenzione : la classe derivata potrebbe non essere stata ancora completamente inizializzata (il suo costruttore verrà eseguito dopo il costruttore della classe base) e questa tecnica è pericolosa (c'è anche un avvertimento StyleCop per questo). Di solito questo è considerato una cattiva pratica.

Generici costruttori statici

Se il tipo su cui è dichiarato il costruttore statico è generico, il costruttore statico verrà chiamato una volta per ogni combinazione univoca di argomenti generici.

```
class Animal<T>
{
    static Animal()
    {
        Console.WriteLine(typeof(T).FullName);
    }

    public static void Yawn() { }
}

Animal<Object>.Yawn();
Animal<String>.Yawn();
```

Questo produrrà:

System.Object
System.String

Vedi anche Come funzionano i costruttori statici per tipi generici?

Eccezioni nei costruttori statici

Se un costruttore statico genera un'eccezione, non viene mai ripetuto. Il tipo è inutilizzabile per la durata di AppDomain. Qualsiasi ulteriore utilizzo del tipo genererà una `TypeInitializationException` racchiusa `TypeInitializationException` originale.

```
public class Animal
{
    static Animal()
    {
        Console.WriteLine("Static ctor");
        throw new Exception();
    }

    public static void Yawn() {}
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}
```

Questo produrrà:

> Agente statico
>
> System.TypeInitializationException: l'inizializzatore del tipo per 'Animal' ha generato un'eccezione. ---> System.Exception: è stata generata un'eccezione di tipo "System.Exception".

[...]

> System.TypeInitializationException: l'inizializzatore del tipo per 'Animal' ha generato un'eccezione. ---> System.Exception: è stata generata un'eccezione di tipo "System.Exception".

dove puoi vedere che il costruttore vero e proprio viene eseguito una sola volta e l'eccezione viene riutilizzata.

Inizializzazione costruttore e proprietà

Il compito del valore della proprietà deve essere eseguito *prima* o *dopo* il costruttore della classe?

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }
}

var testInstance = new TestClass() { TestProperty = 1 };
```

Nell'esempio sopra, il valore `TestProperty` deve essere `1` nel costruttore della classe o dopo il costruttore della classe?

Assegnazione dei valori delle proprietà nella creazione dell'istanza in questo modo:

```
var testInstance = new TestClass() {TestProperty = 1};
```

Verrà eseguito ***dopo*** l'esecuzione del costruttore. Tuttavia, inizializzando il valore della proprietà nella proprietà della classe in C # 6.0 in questo modo:

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
    }
}
```

sarà fatto ***prima*** che venga eseguito il costruttore.

Combinando i due concetti sopra in un unico esempio:

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
```

```
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }
}

static void Main(string[] args)
{
    var testInstance = new TestClass() { TestProperty = 1 };
    Console.WriteLine(testInstance.TestProperty); //resulting in 1
}
```

Risultato finale:

```
"Or shall this be executed"
"1"
```

Spiegazione:

Il valore `TestProperty` verrà prima assegnato come `2` , quindi verrà eseguito il costruttore `TestClass` , con conseguente stampa di

```
"Or shall this be executed"
```

E quindi `TestProperty` verrà assegnato come `1` causa del `new TestClass() { TestProperty = 1 }` , rendendo il valore finale per `TestProperty` stampato da `Console.WriteLine(testInstance.TestProperty)` da

```
"1"
```

Leggi Costruttori e Finalizzatori online: https://riptutorial.com/it/csharp/topic/25/costruttori-e-finalizzatori

Capitolo 38: Creazione del proprio MessageBox nell'applicazione Windows Form

introduzione

Per prima cosa dobbiamo sapere cos'è un MessageBox ...

Il controllo MessageBox visualizza un messaggio con testo specificato e può essere personalizzato specificando un'immagine personalizzata, titolo e set di pulsanti (questi set di pulsanti consentono all'utente di scegliere più di una semplice risposta sì / no).

Creando il nostro MessageBox, possiamo riutilizzare il controllo MessageBox in qualsiasi nuova applicazione semplicemente utilizzando la DLL generata o copiando il file che contiene la classe.

Sintassi

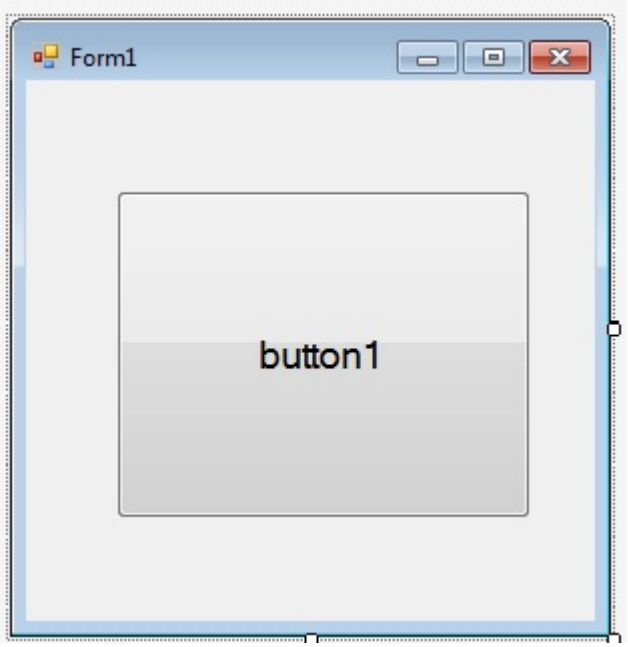
- 'static DialogResult result = DialogResult.No; // DialogResult viene restituito dalle finestre di dialogo dopo il licenziamento. '

Examples

Creazione del proprio controllo MessageBox.

Per creare il nostro controllo MessageBox basta seguire la guida qui sotto ...

1. Apri la tua istanza di Visual Studio (VS 2008/2010/2012/2015/2017)
2. Vai alla barra degli strumenti in alto e fai clic su File -> Nuovo progetto -> Applicazione Windows Form -> Assegna un nome al progetto, quindi fai clic su OK.
3. Una volta caricato, trascinare e rilasciare un pulsante di controllo dalla casella degli strumenti (che si trova a sinistra) sul modulo (come mostrato di seguito).



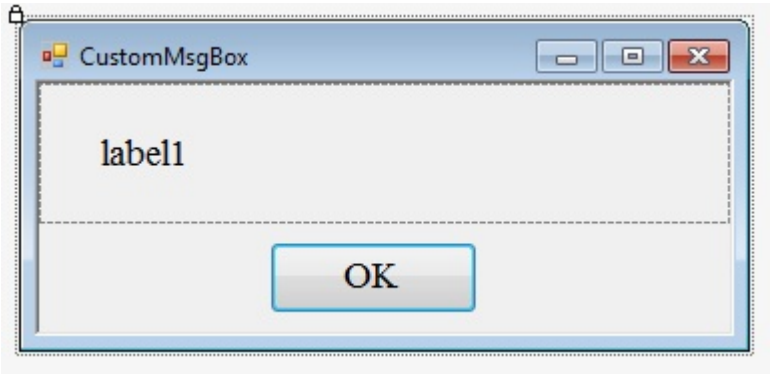
4. Fare doppio clic sul pulsante e l'Integrated Development Environment genererà automaticamente il gestore di eventi click per te.

5. Modificare il codice per il modulo in modo che assomigli al seguente (è possibile fare clic con il pulsante destro del mouse sul modulo e fare clic su Modifica codice):

```
namespace MsgBoxExample {
    public partial class MsgBoxExampleForm : Form {
        //Constructor, called when the class is initialised.
        public MsgBoxExampleForm() {
            InitializeComponent();
        }

        //Called whenever the button is clicked.
        private void btnShowMessageBox_Click(object sender, EventArgs e) {
           CustomMsgBox.Show($"I'm a {nameof(CustomMsgBox)}!", "MSG", "OK");
        }
    }
}
```

6. Solution Explorer -> Fare clic con il tasto destro sul progetto -> Aggiungi -> Windows Form e impostare il nome come "CustomMsgBox.cs"

7. Trascina un pulsante e un controllo etichetta dalla casella degli strumenti al modulo (dopo averlo visualizzato sarà simile al modulo seguente:



8. Ora scrivi il codice qui sotto nel modulo appena creato:

```
private DialogResult result = DialogResult.No;
public static DialogResult Show(string text, string caption, string btnOkText) {
    var msgBox = new CustomMsgBox();
    msgBox.lblText.Text = text; //The text for the label...
    msgBox.Text = caption; //Title of form
    msgBox.btnOk.Text = btnOkText; //Text on the button
    //This method is blocking, and will only return once the user
    //clicks ok or closes the form.
    msgBox.ShowDialog();
    return result;
}

private void btnOk_Click(object sender, EventArgs e) {
    result = DialogResult.Yes;
    MsgBox.Close();
}
```

9. Ora esegui il programma premendo il tasto F5. Congratulazioni, hai fatto un controllo riutilizzabile.

Come utilizzare il proprio controllo MessageBox creato in un'altra applicazione Windows Form.

Per trovare i file .cs esistenti, fare clic con il tasto destro del mouse sul progetto nell'istanza di Visual Studio e fare clic su Apri cartella in Esplora file.

1. Visual Studio -> Il tuo progetto corrente (Windows Form) -> Esplora soluzioni -> Nome progetto -> Clic destro -> Aggiungi -> Oggetto esistente -> Quindi individua il tuo file .cs esistente.

2. Ora c'è un'ultima cosa da fare per usare il controllo. Aggiungi un'istruzione using al tuo codice, in modo che l'assembly possa conoscere le sue dipendenze.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
.
.
```

```
.
using CustomMsgBox; //Here's the using statement for our dependency.
```

3. Per visualizzare la messagebox, usa semplicemente la seguente ...

   CustomMsgBox.Show ("Il tuo messaggio per Message Box ...", "MSG", "OK");

Leggi Creazione del proprio MessageBox nell'applicazione Windows Form online: https://riptutorial.com/it/csharp/topic/9788/creazione-del-proprio-messagebox-nell-applicazione-windows-form

Capitolo 39: Creazione di un'applicazione console utilizzando un Editor di testo semplice e il compilatore C # (csc.exe)

Examples

Creazione di un'applicazione console utilizzando un Editor di testo semplice e il compilatore C

Per utilizzare un editor di testo semplice per creare un'applicazione console scritta in C #, è necessario il compilatore C #. Il compilatore C # (csc.exe) può essere trovato nel seguente percorso: `%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe`

NB In base alla versione di .NET Framework installata nel sistema, potrebbe essere necessario modificare di conseguenza il percorso sopra riportato.

Salvataggio del codice

Lo scopo di questo tema non è quello di insegnare *come* scrivere un'applicazione console, ma per insegnarvi come *compilare* un [per produrre un unico file eseguibile], con niente altro che il compilatore C # e qualsiasi editor di testo (ad esempio, Bloc notes).

1. Apri la finestra di dialogo Esegui, utilizzando la scorciatoia da tastiera `Tasto Windows` + `R`
2. Digita il `notepad` , quindi premi `Invio`
3. Incolla il codice di esempio qui sotto, nel Blocco note
4. Salvare il file come `ConsoleApp.cs` , andando su **File** → **Salva come ...** , quindi inserendo `ConsoleApp.cs` nel campo di testo 'Nome file', quindi selezionando `All Files` come tipo di file.
5. Fai `Save` su `Save`

Compilare il codice sorgente

1. Aprire la finestra di dialogo Esegui, utilizzando il `tasto Windows` + `R`
2. Immettere:

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe
/out:"C:\Users\yourUserName\Documents\ConsoleApp.exe"
"C:\Users\yourUserName\Documents\ConsoleApp.cs"
```

Ora, torna al punto in cui hai salvato in origine il tuo file `ConsoleApp.cs` . Ora dovresti vedere un file

eseguibile (`ConsoleApp.exe`). Fare doppio clic su `ConsoleApp.exe` per aprirlo.

Questo è tutto! La tua applicazione per la console è stata compilata. È stato creato un file eseguibile e ora hai un'app Console funzionante.

```
using System;

namespace ConsoleApp
{
    class Program
    {
        private static string input = String.Empty;

        static void Main(string[] args)
        {
            goto DisplayGreeting;

            DisplayGreeting:
            {
                Console.WriteLine("Hello! What is your name?");

                input = Console.ReadLine();

                if (input.Length >= 1)
                {
                    Console.WriteLine(
                        "Hello, " +
                        input +
                        ", enter 'Exit' at any time to exit this app.");

                    goto AwaitFurtherInstruction;
                }
                else
                {
                    goto DisplayGreeting;
                }
            }

            AwaitFurtherInstruction:
            {
                input = Console.ReadLine();

                if(input.ToLower() == "exit")
                {
                    input = String.Empty;

                    Environment.Exit(0);
                }
                else
                {
                    goto AwaitFurtherInstruction;
                }
            }
        }
    }
}
```

Leggi Creazione di un'applicazione console utilizzando un Editor di testo semplice e il compilatore C # (csc.exe) online: https://riptutorial.com/it/csharp/topic/6676/creazione-di-un-applicazione-

console-utilizzando-un-editor-di-testo-semplice-e-il-compilatore-c-sharp--csc-exe-

Capitolo 40: Crittografia (System.Security.Cryptography)

Examples

Esempi moderni di crittografia autenticata simmetrica di una stringa

La crittografia è qualcosa di molto difficile e dopo aver passato molto tempo a leggere diversi esempi e vedere quanto sia facile introdurre una qualche forma di vulnerabilità ho trovato una risposta originariamente scritta da @jbtule che ritengo sia molto buona. Buona lettura:

"La migliore medicina generale per la crittografia simmetrica è quello di utilizzare la crittografia autenticato con Associated Data (AEAD), tuttavia questo non è una parte delle librerie crittografiche .net standard. Quindi il primo esempio utilizza AES256 e poi HMAC256 , a due step Encrypt poi MAC , che richiede più overhead e più chiavi.

Il secondo esempio utilizza la pratica più semplice di AES256- GCM utilizzando il castello di Bouncy open source (via nuget).

Entrambi gli esempi hanno una funzione principale che accetta una stringa di messaggio segreta, una o più chiavi e un payload e una stringa crittografati facoltativi e non crittografati facoltativamente preceduti dai dati non segreti. Idealmente `NewKey()` questi con chiavi a 256 bit generate casualmente, vedi `NewKey()` .

Entrambi gli esempi hanno anche metodi di supporto che utilizzano una password di stringa per generare le chiavi. Questi metodi di supporto sono forniti per comodità con altri esempi, tuttavia sono *molto meno sicuri* perché la forza della password sarà *molto più debole di una chiave a 256 bit* .

Aggiornamento: sovraccarico di `byte[]` aggiunto, e solo Gist ha la formattazione completa con rientri a 4 spazi e documenti API a causa dei limiti di risposta di StackOverflow. "

.NET Built-in Encrypt (AES) -Then-MAC (HMAC) [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace Encryption
```

```
{
  public static class AESThenHMAC
  {
    private static readonly RandomNumberGenerator Random = RandomNumberGenerator.Create();

    //Preconfigured Encryption Parameters
    public static readonly int BlockBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 64;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;

    /// <summary>
    /// Helper that generates a random key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.GetBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) for a UTF8 Message.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayload">(Optional) Non-Secret Payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message-Padded-To-Blocksize +
HMac-Tag(32)) * 1.33 Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] cryptKey, byte[] authKey,
                        byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, cryptKey, authKey, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) then Decryption (AES) for a secrets UTF8 Message.
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
```

```
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    public static string SimpleDecrypt(string encryptedMessage, byte[] cryptKey, byte[]
authKey,
                        int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, cryptKey, authKey, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) of a UTF8 message
    /// using Keys derived from a Password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">password</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) and then Descryption (AES) of a UTF8 Message
    /// using keys derived from a password (PBKDF2).
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");
```

```
      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] cryptKey, byte[] authKey,
byte[] nonSecretPayload = null)
    {
      //User Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"authKey");

      if (secretMessage == null || secretMessage.Length < 1)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //non-secret payload optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      byte[] cipherText;
      byte[] iv;

      using (var aes = new AesManaged
      {
        KeySize = KeyBitSize,
        BlockSize = BlockBitSize,
        Mode = CipherMode.CBC,
        Padding = PaddingMode.PKCS7
      })
      {

        //Use random IV
        aes.GenerateIV();
        iv = aes.IV;

        using (var encrypter = aes.CreateEncryptor(cryptKey, iv))
        using (var cipherStream = new MemoryStream())
        {
          using (var cryptoStream = new CryptoStream(cipherStream, encrypter,
CryptoStreamMode.Write))
          using (var binaryWriter = new BinaryWriter(cryptoStream))
          {
            //Encrypt Data
            binaryWriter.Write(secretMessage);
          }

          cipherText = cipherStream.ToArray();
        }

      }

      //Assemble encrypted message and add authentication
      using (var hmac = new HMACSHA256(authKey))
      using (var encryptedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(encryptedStream))
```

```
        {
          //Prepend non-secret payload if any
          binaryWriter.Write(nonSecretPayload);
          //Prepend IV
          binaryWriter.Write(iv);
          //Write Ciphertext
          binaryWriter.Write(cipherText);
          binaryWriter.Flush();

          //Authenticate all data
          var tag = hmac.ComputeHash(encryptedStream.ToArray());
          //Postpend tag
          binaryWriter.Write(tag);
        }
        return encryptedStream.ToArray();
      }

    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] cryptKey, byte[]
authKey, int nonSecretPayloadLength = 0)
    {

      //Basic Usage Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("CryptKey needs to be {0} bit!",
KeyBitSize), "cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("AuthKey needs to be {0} bit!", KeyBitSize),
"authKey");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var hmac = new HMACSHA256(authKey))
      {
        var sentTag = new byte[hmac.HashSize / 8];
        //Calculate Tag
        var calcTag = hmac.ComputeHash(encryptedMessage, 0, encryptedMessage.Length -
sentTag.Length);
        var ivLength = (BlockBitSize / 8);

        //if message length is to small just return null
        if (encryptedMessage.Length < sentTag.Length + nonSecretPayloadLength + ivLength)
          return null;

        //Grab Sent Tag
        Array.Copy(encryptedMessage, encryptedMessage.Length - sentTag.Length, sentTag, 0,
sentTag.Length);

        //Compare Tag with constant time comparison
        var compare = 0;
        for (var i = 0; i < sentTag.Length; i++)
          compare |= sentTag[i] ^ calcTag[i];

        //if message doesn't authenticate return null
        if (compare != 0)
          return null;

        using (var aes = new AesManaged
```

```
        {
          KeySize = KeyBitSize,
          BlockSize = BlockBitSize,
          Mode = CipherMode.CBC,
          Padding = PaddingMode.PKCS7
        })
        {

          //Grab IV from message
          var iv = new byte[ivLength];
          Array.Copy(encryptedMessage, nonSecretPayloadLength, iv, 0, iv.Length);

          using (var decrypter = aes.CreateDecryptor(cryptKey, iv))
          using (var plainTextStream = new MemoryStream())
          {
            using (var decrypterStream = new CryptoStream(plainTextStream, decrypter,
CryptoStreamMode.Write))
            using (var binaryWriter = new BinaryWriter(decrypterStream))
            {
              //Decrypt Cipher Text from Message
              binaryWriter.Write(
                encryptedMessage,
                nonSecretPayloadLength + iv.Length,
                encryptedMessage.Length - nonSecretPayloadLength - iv.Length - sentTag.Length
              );
            }
            //Return Plain Text
            return plainTextStream.ToArray();
          }
        }
      }
    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length ==0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var payload = new byte[((SaltBitSize / 8) * 2) + nonSecretPayload.Length];

      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      int payloadIndex = nonSecretPayload.Length;

      byte[] cryptKey;
      byte[] authKey;
      //Use Random Salt to prevent pre-generated weak password attacks.
      using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
      {
        var salt = generator.Salt;

        //Generate Keys
        cryptKey = generator.GetBytes(KeyBitSize / 8);
```

```
        //Create Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
        payloadIndex += salt.Length;
      }

      //Deriving separate key, might be less efficient than using HKDF,
      //but now compatible with RNEncryptor which had a very similar wireformat and requires
less code than HKDF.
      using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
      {
        var salt = generator.Salt;

        //Generate Keys
        authKey = generator.GetBytes(KeyBitSize / 8);

        //Create Rest of Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
      }

      return SimpleEncrypt(secretMessage, cryptKey, authKey, payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cryptSalt = new byte[SaltBitSize / 8];
      var authSalt = new byte[SaltBitSize / 8];

      //Grab Salt from Non-Secret Payload
      Array.Copy(encryptedMessage, nonSecretPayloadLength, cryptSalt, 0, cryptSalt.Length);
      Array.Copy(encryptedMessage, nonSecretPayloadLength + cryptSalt.Length, authSalt, 0,
authSalt.Length);

      byte[] cryptKey;
      byte[] authKey;

      //Generate crypt key
      using (var generator = new Rfc2898DeriveBytes(password, cryptSalt, Iterations))
      {
        cryptKey = generator.GetBytes(KeyBitSize / 8);
      }
      //Generate auth key
      using (var generator = new Rfc2898DeriveBytes(password, authSalt, Iterations))
      {
        authKey = generator.GetBytes(KeyBitSize / 8);
      }

      return SimpleDecrypt(encryptedMessage, cryptKey, authKey, cryptSalt.Length +
authSalt.Length + nonSecretPayloadLength);
    }
  }
}
```

Castello gonfiabile AES-GCM [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Text;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Crypto.Engines;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Modes;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Security;
namespace Encryption
{

  public static class AESGCM
  {
    private static readonly SecureRandom Random = new SecureRandom();

    //Preconfigured Encryption Parameters
    public static readonly int NonceBitSize = 128;
    public static readonly int MacBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 128;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;


    /// <summary>
    /// Helper that generates a random new key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.NextBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 string.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayload">Optional non-secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message
Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message +  HMac-Tag(16)) * 1.33
Base64
```

```
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] key, byte[]
nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, key, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption & Authentication (AES-GCM) of a UTF8 Message
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayloadLength">Length of the optional non-secret
payload.</param>
    /// <returns>Decrypted Message</returns>
    public static string SimpleDecrypt(string encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrEmpty(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, key, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 String
    /// using key derived from a password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption and Authentication (AES-GCM) of a UTF8 message
    /// using a key derived from a password (PBKDF2)
```

```
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] key, byte[]
nonSecretPayload = null)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //Non-secret Payload Optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      //Using random nonce large enough not to repeat
      var nonce = new byte[NonceBitSize / 8];
      Random.NextBytes(nonce, 0, nonce.Length);

      var cipher = new GcmBlockCipher(new AesFastEngine());
      var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
      cipher.Init(true, parameters);

      //Generate Cipher Text With Auth Tag
      var cipherText = new byte[cipher.GetOutputSize(secretMessage.Length)];
      var len = cipher.ProcessBytes(secretMessage, 0, secretMessage.Length, cipherText, 0);
      cipher.DoFinal(cipherText, len);

      //Assemble Message
      using (var combinedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(combinedStream))
        {
          //Prepend Authenticated Payload
          binaryWriter.Write(nonSecretPayload);
          //Prepend Nonce
          binaryWriter.Write(nonce);
          //Write Cipher Text
```

```
          binaryWriter.Write(cipherText);
        }
        return combinedStream.ToArray();
      }
    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"key");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var cipherStream = new MemoryStream(encryptedMessage))
      using (var cipherReader = new BinaryReader(cipherStream))
      {
        //Grab Payload
        var nonSecretPayload = cipherReader.ReadBytes(nonSecretPayloadLength);

        //Grab Nonce
        var nonce = cipherReader.ReadBytes(NonceBitSize / 8);

        var cipher = new GcmBlockCipher(new AesFastEngine());
        var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
        cipher.Init(false, parameters);

        //Decrypt Cipher Text
        var cipherText = cipherReader.ReadBytes(encryptedMessage.Length -
nonSecretPayloadLength - nonce.Length);
        var plainText = new byte[cipher.GetOutputSize(cipherText.Length)];

        try
        {
          var len = cipher.ProcessBytes(cipherText, 0, cipherText.Length, plainText, 0);
          cipher.DoFinal(plainText, len);

        }
        catch (InvalidCipherTextException)
        {
          //Return null if it doesn't authenticate
          return null;
        }

        return plainText;
      }

    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password,
byte[] nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
```

```
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Use Random Salt to minimize pre-generated weak password attacks.
      var salt = new byte[SaltBitSize / 8];
      Random.NextBytes(salt);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      //Create Full Non Secret Payload
      var payload = new byte[salt.Length + nonSecretPayload.Length];
      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      Array.Copy(salt,0, payload,nonSecretPayload.Length, salt.Length);

      return SimpleEncrypt(secretMessage, key.GetKey(), payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password,
int nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Grab Salt from Payload
      var salt = new byte[SaltBitSize / 8];
      Array.Copy(encryptedMessage, nonSecretPayloadLength, salt, 0, salt.Length);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      return SimpleDecrypt(encryptedMessage, key.GetKey(), salt.Length +
nonSecretPayloadLength);
    }
  }
}
```

Introduzione alla crittografia simmetrica e asimmetrica

È possibile migliorare la sicurezza per il transito o l'archiviazione dei dati implementando tecniche di crittografia. Fondamentalmente ci sono due approcci quando si utilizza *System.Security.Cryptography* : **simmetrica** e **asimmetrica.**

Crittografia simmetrica

Questo metodo utilizza una chiave privata per eseguire la trasformazione dei dati.

Professionisti:

- Gli algoritmi simmetrici consumano meno risorse e sono più veloci di quelli asimmetrici.
- La quantità di dati che puoi crittografare è illimitata.

Contro:

- La crittografia e la decrittografia utilizzano la stessa chiave. Qualcuno sarà in grado di decrittografare i tuoi dati se la chiave è compromessa.
- Potresti finire con molte chiavi segrete da gestire se scegli di utilizzare una chiave segreta diversa per dati diversi.

In System.Security.Cryptography sono disponibili classi diverse che eseguono la crittografia simmetrica, denominate cifrari a blocchi :

- AesManaged (algoritmo AES).
- AesCryptoServiceProvider (algoritmo AES FIPS 140-2 reclamo).
- DESCryptoServiceProvider (algoritmo DES).
- RC2CryptoServiceProvider (algoritmo Rivest Cipher 2).
- RijndaelManaged (algoritmo AES). *Nota* : RijndaelManaged **non** è un reclamo FIPS-197 .
- TripleDES (algoritmo TripleDES).

Crittografia asimmetrica

Questo metodo utilizza una combinazione di chiavi pubbliche e private per eseguire la trasformazione dei dati.

Professionisti:

- Utilizza chiavi più grandi di algoritmi simmetrici, quindi sono meno suscettibili di essere fessurate usando la forza bruta.
- È più facile garantire chi è in grado di crittografare e decrittografare i dati perché si basa su due chiavi (pubbliche e private).

Contro:

- Esiste un limite alla quantità di dati che è possibile crittografare. Il limite è diverso per ciascun algoritmo ed è in genere proporzionale alla dimensione della chiave dell'algoritmo.

Ad esempio, un oggetto RSACryptoServiceProvider con una lunghezza chiave di 1.024 bit può solo crittografare un messaggio inferiore a 128 byte.
- Gli algoritmi asimmetrici sono molto lenti rispetto agli algoritmi simmetrici.

Sotto System.Security.Cryptography hai accesso a diverse classi che eseguono la crittografia asimmetrica:

- DSACryptoServiceProvider (algoritmo Algoritmo di firma digitale)
- RSACryptoServiceProvider (algoritmo Algoritmo RSA)

Hash password

Le password non dovrebbero mai essere archiviate come testo normale! Dovrebbero essere sottoposti a hash con un sale generato in modo casuale (per difendersi dagli attacchi del rainbow table) usando un algoritmo di hashing della password lenta. Un elevato numero di iterazioni (> 10k) può essere utilizzato per rallentare gli attacchi di forza bruta. Un ritardo di ~ 100 ms è accettabile per un utente che accede, ma rende difficile rompere una password lunga. Quando si sceglie un numero di iterazioni, è necessario utilizzare il valore massimo tollerabile per la propria applicazione e aumentarlo man mano che le prestazioni del computer migliorano. Dovrai inoltre considerare l'interruzione di richieste ripetute che potrebbero essere utilizzate come attacco DoS.

Quando si hashing per la prima volta che un sale può essere generato per te, il hash e il sale risultanti possono quindi essere memorizzati in un file.

```
private void firstHash(string userName, string userPassword, int numberOfItterations)
{
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, 8, numberOfItterations);
//Hash the password with a 8 byte salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    byte[] salt = PBKDF2.Salt;
    writeHashToFile(userName, hashedPassword, salt, numberOfItterations); //Store the hashed
password with the salt and number of itterations to check against future password entries
}
```

Verifica la password di un utente esistente, leggi il suo hash e il sale da un file e confronta con l'hash della password inserita

```
private bool checkPassword(string userName, string userPassword, int numberOfItterations)
{
    byte[] usersHash = getUserHashFromFile(userName);
    byte[] userSalt = getUserSaltFromFile(userName);
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, userSalt,
numberOfItterations);    //Hash the password with the users salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    bool passwordsMach = comparePasswords(usersHash, hashedPassword);    //Compares byte
arrays
    return passwordsMach;
}
```

Semplice crittografia di file simmetrici

Nell'esempio di codice seguente viene illustrato un metodo rapido e semplice per crittografare e decodificare i file utilizzando l'algoritmo di crittografia simmetrica AES.

Il codice genera in modo casuale i vettori Salt e Initialization ogni volta che un file viene crittografato, il che significa che la crittografia dello stesso file con la stessa password porterà sempre a output diversi. Il sale e IV vengono scritti nel file di output in modo che solo la password è necessaria per decrittografarlo.

```
public static void ProcessFile(string inputPath, string password, bool encryptMode, string
outputPath)
{
    using (var cypher = new AesManaged())
    using (var fsIn = new FileStream(inputPath, FileMode.Open))
    using (var fsOut = new FileStream(outputPath, FileMode.Create))
    {
        const int saltLength = 256;
        var salt = new byte[saltLength];
        var iv = new byte[cypher.BlockSize / 8];

        if (encryptMode)
        {
            // Generate random salt and IV, then write them to file
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(salt);
                rng.GetBytes(iv);
            }
            fsOut.Write(salt, 0, salt.Length);
            fsOut.Write(iv, 0, iv.Length);
        }
        else
        {
            // Read the salt and IV from the file
            fsIn.Read(salt, 0, saltLength);
            fsIn.Read(iv, 0, iv.Length);
        }

        // Generate a secure password, based on the password and salt provided
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);

        // Encrypt or decrypt the file
        using (var cryptoTransform = encryptMode
            ? cypher.CreateEncryptor(key, iv)
            : cypher.CreateDecryptor(key, iv))
        using (var cs = new CryptoStream(fsOut, cryptoTransform, CryptoStreamMode.Write))
        {
            fsIn.CopyTo(cs);
        }
    }
}
```

Dati casuali protetti da crittografia

Ci sono volte in cui la classe Random () del framework non può essere considerata abbastanza casuale, dato che si basa su un generatore di numeri psuedo-random. Le classi Crypto del framework, tuttavia, forniscono qualcosa di più robusto sotto forma di RNGCryptoServiceProvider.

I seguenti esempi di codice dimostrano come generare array, stringhe e numeri di byte crittograficamente sicuri.

Matrice di byte casuali

```
public static byte[] GenerateRandomData(int length)
{
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    return rnd;
}
```

Numero intero casuale (con distribuzione uniforme)

```
public static int GenerateRandomInt(int minVal=0, int maxVal=100)
{
    var rnd = new byte[4];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    var i = Math.Abs(BitConverter.ToInt32(rnd, 0));
    return Convert.ToInt32(i % (maxVal - minVal + 1) + minVal);
}
```

Stringa casuale

```
public static string GenerateRandomString(int length, string allowableChars=null)
{
    if (string.IsNullOrEmpty(allowableChars))
        allowableChars = @"ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    // Generate random data
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);

    // Generate the output string
    var allowable = allowableChars.ToCharArray();
    var l = allowable.Length;
    var chars = new char[length];
    for (var i = 0; i < length; i++)
        chars[i] = allowable[rnd[i] % l];

    return new string(chars);
}
```

Crittografia file asimmetrica veloce

La crittografia asimmetrica è spesso considerata preferibile alla crittografia simmetrica per il trasferimento di messaggi ad altre parti. Ciò è dovuto principalmente al fatto che nega molti dei rischi legati allo scambio di una chiave condivisa e garantisce che mentre chiunque abbia la chiave pubblica possa crittografare un messaggio per il destinatario previsto, solo quel destinatario può decrittografarlo. Sfortunatamente il principale aspetto negativo degli algoritmi di crittografia asimmetrica è che sono molto più lenti dei loro cugini simmetrici. Di conseguenza, la

crittografia asimmetrica di file, soprattutto di grandi dimensioni, può spesso essere un processo molto intensivo dal punto di vista computazionale.

Per fornire sicurezza e prestazioni, è possibile adottare un approccio ibrido. Ciò comporta la generazione crittograficamente casuale di un vettore di chiave e di inizializzazione per la crittografia *simmetrica* . Questi valori vengono quindi crittografati utilizzando un algoritmo *asimmetrico* e scritti nel file di output, prima di essere utilizzati per crittografare i dati di origine in modo *simmetrico* e aggiungerli all'output.

Questo approccio fornisce un alto grado di prestazioni e sicurezza, in quanto i dati vengono crittografati utilizzando un algoritmo simmetrico (veloce) e la chiave e iv, entrambi generati casualmente (sicuri) sono crittografati da un algoritmo asimmetrico (sicuro). Ha anche il vantaggio aggiunto che lo stesso payload crittografato in diverse occasioni avrà un testo cifrato molto diverso, poiché le chiavi simmetriche vengono generate casualmente ogni volta.

La seguente classe dimostra la crittografia asimmetrica di stringhe e array di byte, oltre alla crittografia ibrida dei file.

```
public static class AsymmetricProvider
{
    #region Key Generation
    public class KeyPair
    {
        public string PublicKey { get; set; }
        public string PrivateKey { get; set; }
    }

    public static KeyPair GenerateNewKeyPair(int keySize = 4096)
    {
        // KeySize is measured in bits. 1024 is the default, 2048 is better, 4096 is more
robust but takes a fair bit longer to generate.
        using (var rsa = new RSACryptoServiceProvider(keySize))
        {
            return new KeyPair {PublicKey = rsa.ToXmlString(false), PrivateKey =
rsa.ToXmlString(true)};
        }
    }

    #endregion

    #region Asymmetric Data Encryption and Decryption

    public static byte[] EncryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            return asymmetricProvider.Encrypt(data, true);
        }
    }

    public static byte[] DecryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
```

```
            if (asymmetricProvider.PublicOnly)
                throw new Exception("The key provided is a public key and does not contain the
private key elements required for decryption");
            return asymmetricProvider.Decrypt(data, true);
        }
    }

    public static string EncryptString(string value, string publicKey)
    {
        return Convert.ToBase64String(EncryptData(Encoding.UTF8.GetBytes(value), publicKey));
    }

    public static string DecryptString(string value, string privateKey)
    {
        return Encoding.UTF8.GetString(EncryptData(Convert.FromBase64String(value),
privateKey));
    }

    #endregion

    #region Hybrid File Encryption and Decription

    public static void EncryptFile(string inputFilePath, string outputFilePath, string
publicKey)
    {
        using (var symmetricCypher = new AesManaged())
        {
            // Generate random key and IV for symmetric encryption
            var key = new byte[symmetricCypher.KeySize / 8];
            var iv = new byte[symmetricCypher.BlockSize / 8];
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(key);
                rng.GetBytes(iv);
            }

            // Encrypt the symmetric key and IV
            var buf = new byte[key.Length + iv.Length];
            Array.Copy(key, buf, key.Length);
            Array.Copy(iv, 0, buf, key.Length, iv.Length);
            buf = EncryptData(buf, publicKey);

            var bufLen = BitConverter.GetBytes(buf.Length);

            // Symmetrically encrypt the data and write it to the file, along with the
encrypted key and iv
            using (var cypherKey = symmetricCypher.CreateEncryptor(key, iv))
            using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
            using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
            using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
            {
                fsOut.Write(bufLen,0, bufLen.Length);
                fsOut.Write(buf, 0, buf.Length);
                fsIn.CopyTo(cs);
            }
        }
    }

    public static void DecryptFile(string inputFilePath, string outputFilePath, string
privateKey)
    {
```

```
        using (var symmetricCypher = new AesManaged())
        using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
        {
            // Determine the length of the encrypted key and IV
            var buf = new byte[sizeof(int)];
            fsIn.Read(buf, 0, buf.Length);
            var bufLen = BitConverter.ToInt32(buf, 0);

            // Read the encrypted key and IV data from the file and decrypt using the
asymmetric algorithm
            buf = new byte[bufLen];
            fsIn.Read(buf, 0, buf.Length);
            buf = DecryptData(buf, privateKey);

            var key = new byte[symmetricCypher.KeySize / 8];
            var iv = new byte[symmetricCypher.BlockSize / 8];
            Array.Copy(buf, key, key.Length);
            Array.Copy(buf, key.Length, iv, 0, iv.Length);

            // Decript the file data using the symmetric algorithm
            using (var cypherKey = symmetricCypher.CreateDecryptor(key, iv))
            using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
            using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
            {
                fsIn.CopyTo(cs);
            }
        }
    }

    #endregion

    #region Key Storage

    public static void WritePublicKey(string publicKeyFilePath, string publicKey)
    {
        File.WriteAllText(publicKeyFilePath, publicKey);
    }
    public static string ReadPublicKey(string publicKeyFilePath)
    {
        return File.ReadAllText(publicKeyFilePath);
    }

    private const string SymmetricSalt = "Stack_Overflow!"; // Change me!

    public static string ReadPrivateKey(string privateKeyFilePath, string password)
    {
        var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
        var cypherText = File.ReadAllBytes(privateKeyFilePath);

        using (var cypher = new AesManaged())
        {
            var pdb = new Rfc2898DeriveBytes(password, salt);
            var key = pdb.GetBytes(cypher.KeySize / 8);
            var iv = pdb.GetBytes(cypher.BlockSize / 8);

            using (var decryptor = cypher.CreateDecryptor(key, iv))
            using (var msDecrypt = new MemoryStream(cypherText))
            using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read))
            using (var srDecrypt = new StreamReader(csDecrypt))
            {
```

```
                return srDecrypt.ReadToEnd();
            }
        }
    }

    public static void WritePrivateKey(string privateKeyFilePath, string privateKey, string
password)
    {
        var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
        using (var cypher = new AesManaged())
        {
            var pdb = new Rfc2898DeriveBytes(password, salt);
            var key = pdb.GetBytes(cypher.KeySize / 8);
            var iv = pdb.GetBytes(cypher.BlockSize / 8);

            using (var encryptor = cypher.CreateEncryptor(key, iv))
            using (var fsEncrypt = new FileStream(privateKeyFilePath, FileMode.Create))
            using (var csEncrypt = new CryptoStream(fsEncrypt, encryptor,
CryptoStreamMode.Write))
            using (var swEncrypt = new StreamWriter(csEncrypt))
            {
                swEncrypt.Write(privateKey);
            }
        }
    }

    #endregion
}
```

Esempio di utilizzo:

```
private static void HybridCryptoTest(string privateKeyPath, string privateKeyPassword, string
inputPath)
{
    // Setup the test
    var publicKeyPath = Path.ChangeExtension(privateKeyPath, ".public");
    var outputPath = Path.Combine(Path.ChangeExtension(inputPath, ".enc"));
    var testPath = Path.Combine(Path.ChangeExtension(inputPath, ".test"));

    if (!File.Exists(privateKeyPath))
    {
        var keys = AsymmetricProvider.GenerateNewKeyPair(2048);
        AsymmetricProvider.WritePublicKey(publicKeyPath, keys.PublicKey);
        AsymmetricProvider.WritePrivateKey(privateKeyPath, keys.PrivateKey,
privateKeyPassword);
    }

    // Encrypt the file
    var publicKey = AsymmetricProvider.ReadPublicKey(publicKeyPath);
    AsymmetricProvider.EncryptFile(inputPath, outputPath, publicKey);

    // Decrypt it again to compare against the source file
    var privateKey = AsymmetricProvider.ReadPrivateKey(privateKeyPath, privateKeyPassword);
    AsymmetricProvider.DecryptFile(outputPath, testPath, privateKey);

    // Check that the two files match
    var source = File.ReadAllBytes(inputPath);
    var dest = File.ReadAllBytes(testPath);

    if (source.Length != dest.Length)
```

```
        throw new Exception("Length does not match");

    if (source.Where((t, i) => t != dest[i]).Any())
        throw new Exception("Data mismatch");
}
```

Leggi Crittografia (System.Security.Cryptography) online:
https://riptutorial.com/it/csharp/topic/2988/crittografia--system-security-cryptography-

Capitolo 41: Cronometri

Sintassi

- stopWatch.Start () - Avvia il cronometro.
- stopWatch.Stop () - Interrompe il cronometro.
- stopWatch.Elapsed - Ottiene il tempo trascorso totale misurato dall'intervallo corrente.

Osservazioni

I cronometri vengono spesso utilizzati nei programmi di benchmarking per il time code e vengono visualizzati i diversi segmenti di codice ottimali da eseguire.

Examples

Creazione di un'istanza di un cronometro

Un'istanza di cronometro può misurare il tempo trascorso su più intervalli con il tempo trascorso totale, essendo tutti gli intervalli individuali sommati tra loro. Ciò fornisce un metodo affidabile per misurare il tempo trascorso tra due o più eventi.

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

double d = 0;
for (int i = 0; i < 1000 * 1000 * 1000; i++)
{
    d += 1;
}

stopWatch.Stop();
Console.WriteLine("Time elapsed: {0:hh\\:mm\\:ss\\.fffffff}", stopWatch.Elapsed);
```

`Stopwach` è in `System.Diagnostics` quindi è necessario aggiungere `using System.Diagnostics;` al tuo file.

IsHighResolution

- La proprietà IsHighResolution indica se il timer è basato su un contatore di prestazioni ad alta risoluzione o basato sulla classe DateTime.
- Questo campo è di sola lettura.

```
// Display the timer frequency and resolution.
if (Stopwatch.IsHighResolution)
{
    Console.WriteLine("Operations timed using the system's high-resolution performance
counter.");
```

```
}
else
{
    Console.WriteLine("Operations timed using the DateTime class.");
}

long frequency = Stopwatch.Frequency;
Console.WriteLine("  Timer frequency in ticks per second = {0}",
    frequency);
long nanosecPerTick = (1000L*1000L*1000L) / frequency;
Console.WriteLine("  Timer is accurate within {0} nanoseconds",
    nanosecPerTick);
}
```

https://dotnetfiddle.net/ckrWUo

Il timer utilizzato dalla classe Cronometro dipende dall'hardware del sistema e dal sistema operativo. IsHighResolution è true se il timer del cronometro è basato su un contatore di prestazioni ad alta risoluzione. In caso contrario, IsHighResolution è false, il che indica che il timer del cronometro è basato sul timer di sistema.

Le zecche in Cronometro dipendono dalla macchina / dal sistema operativo, quindi non si dovrebbe mai contare sulla razione di zecche del cronometro in secondi per essere uguali tra due sistemi, e possibilmente anche sullo stesso sistema dopo un riavvio. Pertanto, non puoi mai contare sui tick di Cronometro per avere lo stesso intervallo dei tick DateTime / TimeSpan.

Per ottenere l'ora indipendente dal sistema, assicurati di utilizzare le proprietà Cronometro esaurite o Trascorse di millisecondi, che tengono già conto del Cronometro. Frequenza (zecche al secondo).

Il cronometro dovrebbe sempre essere utilizzato su Datetime per i processi di temporizzazione in quanto è più leggero e utilizza Dateime se non può utilizzare un contatore di prestazioni ad alta risoluzione.

fonte

Leggi Cronometri online: https://riptutorial.com/it/csharp/topic/3676/cronometri

Capitolo 42: Diagnostica

Examples

Debug.WriteLine

Scrive sui listener della traccia nella raccolta Listeners quando l'applicazione viene compilata in configurazione di debug.

```
public static void Main(string[] args)
{
    Debug.WriteLine("Hello");
}
```

In Visual Studio o Xamarin Studio questo apparirà nella finestra Output dell'applicazione. Ciò è dovuto alla presenza del listener di traccia predefinito in TraceListenerCollection.

Reindirizzamento dell'output del registro con TraceListeners

È possibile reindirizzare l'output di debug in un file di testo aggiungendo un TextWriterTraceListener alla raccolta Debug.Listeners.

```
public static void Main(string[] args)
{
    TextWriterTraceListener myWriter = new TextWriterTraceListener(@"debug.txt");
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");

    myWriter.Flush();
}
```

È possibile reindirizzare l'output di debug allo stream di un'applicazione di una console utilizzando un ConsoleTraceListener.

```
public static void Main(string[] args)
{
    ConsoleTraceListener myWriter = new ConsoleTraceListener();
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");
}
```

Leggi Diagnostica online: https://riptutorial.com/it/csharp/topic/2147/diagnostica

Capitolo 43: Dichiarazioni condizionali

Examples

Istruzione If-Else

La programmazione in generale richiede spesso una `decision` o una `branch` all'interno del codice per tenere conto di come il codice opera in presenza di input o condizioni differenti. All'interno del linguaggio di programmazione C # (e della maggior parte dei linguaggi di programmazione per questo argomento), il modo più semplice e talvolta più utile di creare un ramo all'interno del programma è attraverso un'istruzione `If-Else` .

Supponiamo di avere un metodo (ovvero una funzione) che accetta un parametro int che rappresenterà un punteggio fino a 100, e il metodo stamperà un messaggio che dice se passiamo o meno.

```
static void PrintPassOrFail(int score)
{
    if (score >= 50) // If score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If score is not greater or equal to 50
    {
        Console.WriteLine("Fail!");
    }
}
```

Osservando questo metodo, si può notare questa linea di codice (`score >= 50`) all'interno dell'istruzione `If` . Questo può essere visto come una condizione `boolean` , dove se la condizione viene valutata uguale a `true` , allora viene eseguito il codice che si trova tra `if { }` .

Ad esempio, se questo metodo è stato chiamato in questo modo: `PrintPassOrFail(60);` , l'output del metodo sarebbe una stampa console che dice ***Pass!*** poiché il valore del parametro 60 è maggiore o uguale a 50.

Tuttavia, se il metodo è stato chiamato come: `PrintPassOrFail(30);` , l'output del metodo verrebbe stampato dicendo ***Fail!*** . Questo perché il valore 30 non è maggiore o uguale a 50, quindi viene eseguito il codice tra `else { }` anziché l'istruzione `If` .

In questo esempio, abbiamo detto che il *punteggio* dovrebbe arrivare a 100, che non è stato affatto calcolato. Per tenere conto del *punteggio che* non va oltre il 100 o eventualmente scendere al di sotto di 0, vedere l'esempio **dell'istruzione If-Else If-Else** .

If-Else If-Else Statement

Seguendo l'esempio **dell'istruzione If-Else** , è giunto il momento di introdurre l'istruzione `Else If` . L'istruzione `Else If` segue direttamente dopo l'istruzione `If` nella struttura **If-Else If-Else** , ma

intrinsecamente ha una sintassi simile all'istruzione `If` . È usato per aggiungere più rami al codice di quello che può fare una semplice istruzione **If-Else** .

Nell'esempio di **If-Else Statement** , l'esempio ha specificato che il punteggio sale a 100; tuttavia non ci sono mai stati controlli contro questo. Per risolvere questo problema, è possibile modificare il metodo dall'istruzione **If-Else** in modo che assomigli a questo:

```
static void PrintPassOrFail(int score)
{
    if (score > 100) // If score is greater than 100
    {
        Console.WriteLine("Error: score is greater than 100!");
    }
    else if (score < 0) // Else If score is less than 0
    {
        Console.WriteLine("Error: score is less than 0!");
    }
    else if (score >= 50) // Else if score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If none above, then score must be between 0 and 49
    {
        Console.WriteLine("Fail!");
    }
}
```

Tutte queste affermazioni funzioneranno in ordine dall'alto verso il basso fino a quando una condizione non sarà soddisfatta. In questo nuovo aggiornamento del metodo, abbiamo aggiunto due nuovi rami per ospitare ora il punteggio in *uscita* .

Ad esempio, se ora chiamiamo il metodo nel nostro codice come `PrintPassOFail(110);` , l'output sarebbe una stampa della console che dice ***Errore: il punteggio è maggiore di 100!*** ; e se abbiamo chiamato il metodo nel nostro codice come `PrintPassOrFail(-20);` , l'output direbbe ***Errore: il punteggio è inferiore a 0!*** .

Cambia istruzioni

Un'istruzione switch consente di verificare una variabile per l'uguaglianza rispetto a un elenco di valori. Ogni valore è chiamato caso e la variabile che viene attivata viene controllata per ogni caso di commutazione.

Un'istruzione `switch` è spesso più concisa e comprensibile rispetto a `if...else if... else..` istruzioni quando si testano più valori possibili per una singola variabile.

La sintassi è la seguente

```
switch(expression) {
   case constant-expression:
      statement(s);
      break;
   case constant-expression:
      statement(s);
```

```
      break;

   // you can have any number of case statements
   default : // Optional
      statement(s);
      break;
}
```

ci sono cose settarie che devono essere considerate mentre si usa l'istruzione switch

- L'espressione utilizzata in un'istruzione switch deve avere un tipo integrale o enumerato oppure essere di un tipo di classe in cui la classe ha una singola funzione di conversione in un tipo integrale o enumerato.
- È possibile avere un numero qualsiasi di istruzioni caso all'interno di un interruttore. Ogni caso è seguito dal valore da confrontare con e dai due punti. I valori da confrontare devono essere unici all'interno di ciascuna istruzione switch.
- Un'istruzione switch può avere un caso predefinito facoltativo. Il caso predefinito può essere utilizzato per eseguire un'attività quando nessuno dei casi è vero.
- Ogni caso deve terminare con un'istruzione `break` meno che non sia un'istruzione vuota. In tal caso l'esecuzione continuerà nel caso sottostante. L'istruzione break può anche essere omessa quando viene utilizzata un'istruzione `goto case` `return` , `throw` o `goto case` .

L'esempio può essere dato con i gradi saggi

```
char grade = 'B';

switch (grade)
{
    case 'A':
        Console.WriteLine("Excellent!");
        break;
    case 'B':
    case 'C':
        Console.WriteLine("Well done");
        break;
    case 'D':
        Console.WriteLine("You passed");
        break;
    case 'F':
        Console.WriteLine("Better try again");
        break;
    default:
        Console.WriteLine("Invalid grade");
        break;
}
```

Se le condizioni dell'istruzione sono espressioni e valori booleani standard

La seguente dichiarazione

```
if (conditionA && conditionB && conditionC) //...
```

è esattamente equivalente a

```
bool conditions = conditionA && conditionB && conditionC;
if (conditions) // ...
```

in altre parole, le condizioni all'interno dell'istruzione "if" formano semplicemente un'espressione booleana ordinaria.

Un errore comune durante la scrittura di istruzioni condizionali è di confrontare esplicitamente con `true` e `false` :

```
if (conditionA == true && conditionB == false && conditionC == true) // ...
```

Questo può essere riscritto come

```
if (conditionA && !conditionB && conditionC)
```

Leggi Dichiarazioni condizionali online: https://riptutorial.com/it/csharp/topic/3144/dichiarazioni-condizionali

Capitolo 44: Direttive preprocessore

Sintassi

- #define *[symbol]* // Definisce un simbolo del compilatore.
- #undef *[symbol]* // Undefines un simbolo del compilatore.
- #warning *[messaggio di avviso]* // Genera un avvertimento del compilatore. Utile con #if.
- #error *[messaggio di errore]* // Genera un errore del compilatore. Utile con #if.
- #line *[numero linea] (nome file)* // Sovrascrive il numero di riga del compilatore (e facoltativamente il nome del file sorgente). Utilizzato con modelli di testo T4 .
- #pragma warning [disable | restore] *[warning numbers]* // Disabilita / ripristina gli avvisi del compilatore.
- #pragma checksum " *[nomefile]* " " *[guid]* " " *[checksum]* " // Convalida il contenuto di un file sorgente.
- #region *[nome regione]* // Definisce una regione di codice comprimibile.
- #endregion // Termina un blocco dell'area del codice.
- #if *[condizione]* // Esegue il codice qui sotto se la condizione è vera.
- #else // Utilizzato dopo un #if.
- #elif *[condizione]* // Utilizzato dopo un #if.
- #endif // Termina un blocco condizionale avviato con #if.

Osservazioni

Le direttive del preprocessore sono generalmente utilizzate per rendere i programmi di origine facili da modificare e facili da compilare in diversi ambienti di esecuzione. Le direttive nel file sorgente indicano al preprocessore di eseguire azioni specifiche. Ad esempio, il preprocessore può sostituire i token nel testo, inserire il contenuto di altri file nel file sorgente o sopprimere la compilazione di parte del file rimuovendo sezioni di testo. Le linee del preprocessore vengono riconosciute e eseguite prima dell'espansione macro. Pertanto, se una macro si espande in qualcosa che assomiglia a un comando del preprocessore, tale comando non viene riconosciuto dal preprocessore.

Le istruzioni di preprocessore utilizzano lo stesso set di caratteri delle istruzioni del file di origine, con l'eccezione che le sequenze di escape non sono supportate. Il set di caratteri utilizzato nelle istruzioni del preprocessore è uguale al set di caratteri di esecuzione. Il preprocessore riconosce anche i valori dei caratteri negativi.

Espressioni condizionali

Le espressioni condizionali (`#if` , `#elif` , ecc.) Supportano un sottoinsieme limitato di operatori booleani. Loro sono:

- `==` e `!=` . Questi possono essere utilizzati solo per verificare se il simbolo è vero (definito) o falso (non definito)
- `&&`

`,` `||` `,` `!`
- `()`

Per esempio:

```
#if !DEBUG && (SOME_SYMBOL || SOME_OTHER_SYMBOL) && RELEASE == true
Console.WriteLine("OK!");
#endif
```

compilerebbe il codice che stampa "OK!" alla console se `DEBUG` non è definito, sono definiti `SOME_SYMBOL` o `SOME_OTHER_SYMBOL` e `RELEASE` è definito.

Nota: queste sostituzioni vengono eseguite *in fase di compilazione* e pertanto non sono disponibili per l'ispezione in fase di esecuzione. Il codice eliminato tramite l'uso di `#if` non fa parte dell'output del compilatore.

Vedere anche: Direttive preprocessore C # a MSDN.

Examples

Espressioni condizionali

Quando viene compilato quanto segue, verrà restituito un valore diverso a seconda delle direttive definite.

```
// Compile with /d:A or /d:B to see the difference
string SomeFunction()
{
#if A
    return "A";
#elif B
    return "B";
#else
    return "C";
#endif
}
```

Le espressioni condizionali vengono in genere utilizzate per registrare informazioni aggiuntive per le build di debug.

```
void SomeFunc()
{
    try
    {
        SomeRiskyMethod();
    }
    catch (ArgumentException ex)
    {
        #if DEBUG
        log.Error("SomeFunc", ex);
        #endif

        HandleException(ex);
```

```
    }
}
```

Generazione di avvisi ed errori del compilatore

Gli avvertimenti del compilatore possono essere generati usando la direttiva `#warning` e gli errori possono essere generati anche usando la direttiva `#error` .

```
#if SOME_SYMBOL
#error This is a compiler Error.
#elif SOME_OTHER_SYMBOL
#warning This is a compiler Warning.
#endif
```

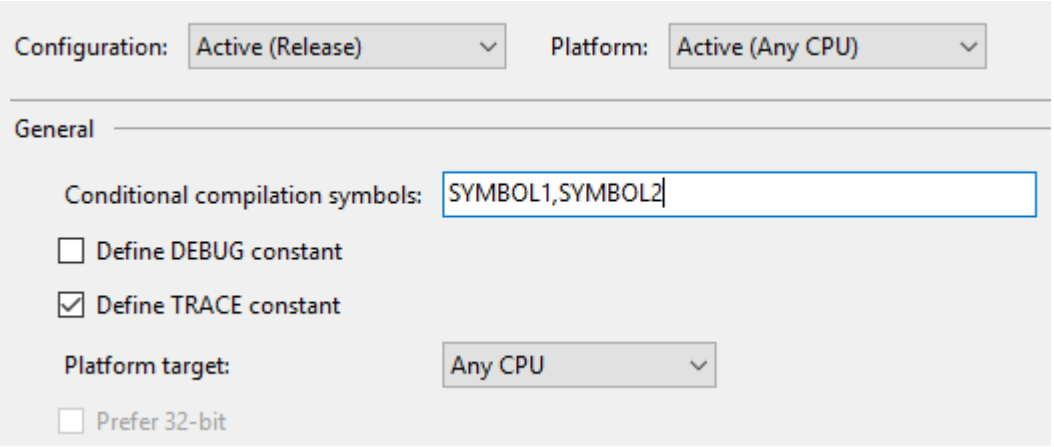
Definizione e annullamento della definizione dei simboli

Un simbolo del compilatore è una parola chiave definita in fase di compilazione che può essere controllata per eseguire in modo condizionato specifiche sezioni di codice.

Esistono tre modi per definire un simbolo del compilatore. Possono essere definiti tramite codice:

```
#define MYSYMBOL
```

Possono essere definiti in Visual Studio, in Proprietà progetto> Crea> Simboli di compilazione condizionale:



(Si noti che `DEBUG` e `TRACE` hanno le proprie caselle di controllo e non è necessario specificare esplicitamente.)

Oppure possono essere definiti in fase di compilazione usando l' `csc.exe /define:[name]` sul compilatore C #, `csc.exe` .

Puoi anche simboli indefiniti usando la direttiva `#undefine` .

L'esempio più diffuso di questo è il simbolo `DEBUG` , che viene definito da Visual Studio quando un'applicazione viene compilata in modalità Debug (rispetto alla modalità di rilascio).

```
public void DoBusinessLogic()
{
    try
    {
        AuthenticateUser();
        LoadAccount();
        ProcessAccount();
        FinalizeTransaction();
    }
    catch (Exception ex)
    {
#if DEBUG
        System.Diagnostics.Trace.WriteLine("Unhandled exception!");
        System.Diagnostics.Trace.WriteLine(ex);
        throw;
#else
        LoggingFramework.LogError(ex);
        DisplayFriendlyErrorMessage();
#endif
    }
}
```

Nell'esempio sopra, quando si verifica un errore nella business logic dell'applicazione, se l'applicazione è compilata in modalità Debug (e il simbolo `DEBUG` è impostato), l'errore verrà scritto nel log di traccia e l'eccezione verrà reimpostata generato per il debug. Tuttavia, se l'applicazione è compilata in modalità Release (e non viene impostato alcun simbolo `DEBUG`), viene utilizzato un framework di registrazione per registrare l'errore in modo silenzioso e viene visualizzato un messaggio di errore amichevole all'utente finale.

Blocchi Regionali

Utilizza `#region` e `#endregion` per definire una regione di codice comprimibile.

```
#region Event Handlers

public void Button_Click(object s, EventArgs e)
{
    // ...
}

public void DropDown_SelectedIndexChanged(object s, EventArgs e)
{
    // ...
}

#endregion
```

Queste direttive sono utili solo quando un IDE che supporta regioni comprimibili (come Visual Studio) viene utilizzato per modificare il codice.

Altre istruzioni per il compilatore

Linea

`#line` controlla il numero di riga e il nome del file riportati dal compilatore durante l'emissione di avvisi ed errori.

```
void Test()
{
    #line 42 "Answer"
    #line filename "SomeFile.cs"
    int life; // compiler warning CS0168 in "SomeFile.cs" at Line 42
    #line default
    // compiler warnings reset to default
}
```

Checksum di Pragma

`#pragma checksum` consente la specifica di un checksum specifico per un database di programma generato (PDB) per il debug.

```
#pragma checksum "MyCode.cs" "{00000000-0000-0000-0000-000000000000}" "{0123456789A}"
```

Utilizzando l'attributo condizionale

Aggiungere un attributo `Conditional` dallo spazio dei nomi `System.Diagnostics` a un metodo è un modo pulito per controllare quali metodi vengono chiamati nelle build e quali no.

```
#define EXAMPLE_A

using System.Diagnostics;
class Program
{
    static void Main()
    {
        ExampleA(); // This method will be called
        ExampleB(); // This method will not be called
    }

    [Conditional("EXAMPLE_A")]
    static void ExampleA() {...}

    [Conditional("EXAMPLE_B")]
    static void ExampleB() {...}
}
```

Disattivazione e ripristino degli avvisi del compilatore

È possibile disabilitare gli avvisi del compilatore utilizzando l' `#pragma warning disable` e ripristinarli utilizzando il `#pragma warning restore` :

```
#pragma warning disable CS0168

// Will not generate the "unused variable" compiler warning since it was disabled
var x = 5;
```

```
#pragma warning restore CS0168

// Will generate a compiler warning since the warning was just restored
var y = 8;
```

Sono ammessi numeri di avviso separati da virgola:

```
#pragma warning disable CS0168, CS0219
```

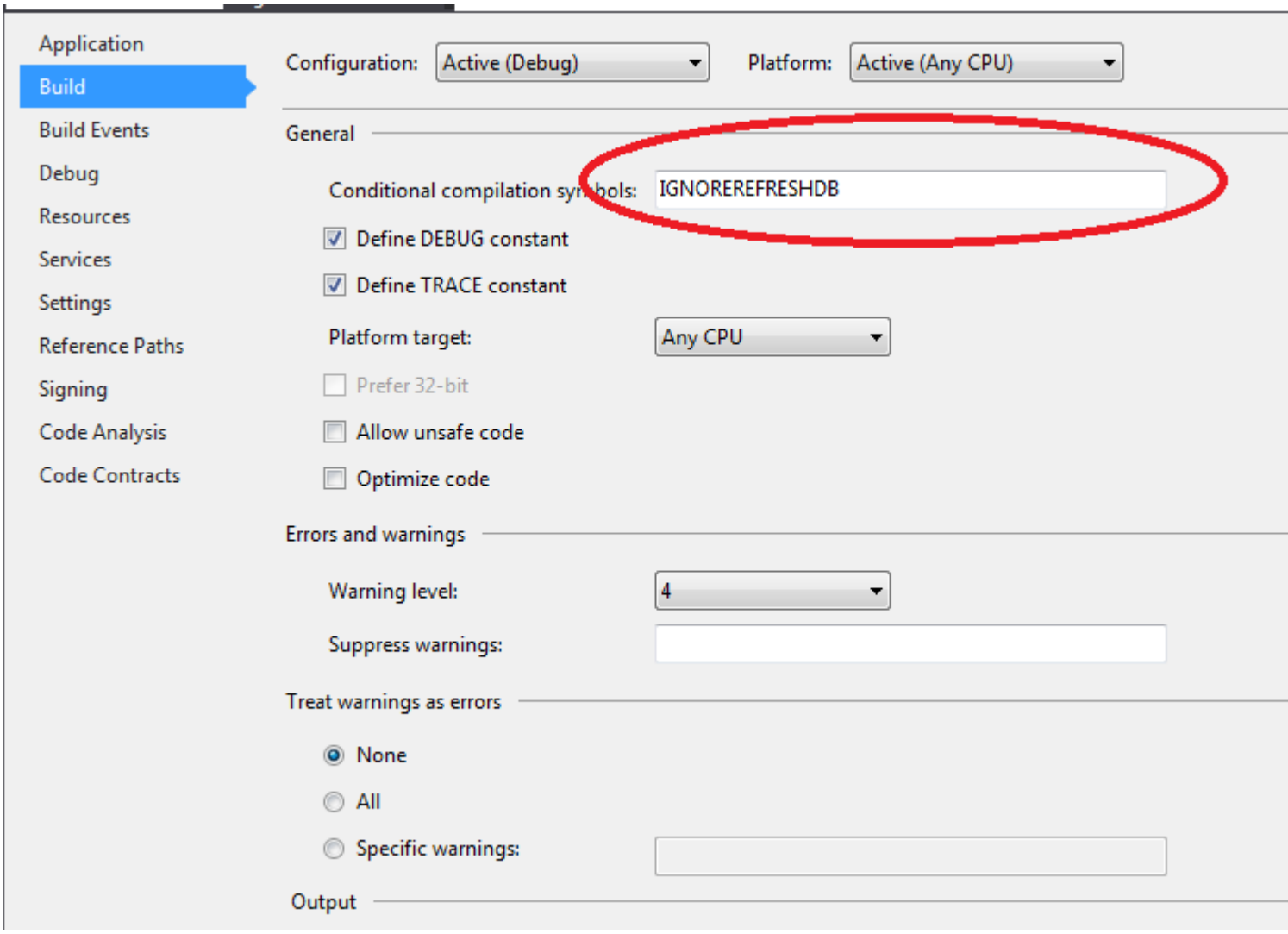
Il prefisso `CS` è facoltativo e può anche essere mescolato (anche se questa non è una best practice):

```
#pragma warning disable 0168, 0219, CS0414
```

Preprocessori personalizzati a livello di progetto

È utile impostare la pre-elaborazione condizionale personalizzata a livello di progetto quando alcune azioni devono essere saltate, diciamo per i test.

Vai a `Solution Explorer` -> Fai clic con il `pulsante destro del mouse` sul progetto per cui vuoi impostare la variabile su -> `Properties` -> `Build` -> Nel campo di ricerca Generale `Conditional compilation symbols` e inserisci qui la variabile condizionale



Esempio di codice che salterà del codice:

```
public void Init()
{
    #if !IGNOREREFRESHDB
    // will skip code here
     db.Initialize();
    #endif
}
```

Leggi Direttive preprocessore online: https://riptutorial.com/it/csharp/topic/755/direttive-preprocessore

Capitolo 45: enum

introduzione

Un enum può derivare da uno dei seguenti tipi: byte, sbyte, short, ushort, int, uint, long, ulong. Il valore predefinito è int e può essere modificato specificando il tipo nella definizione enum:

public enum Weekday: byte {lunedì = 1, martedì = 2, mercoledì = 3, giovedì = 4, venerdì = 5}

Questo è utile quando P / Invoca codice nativo, mappatura a origini dati e circostanze simili. In generale, si dovrebbe usare l'int predefinito, perché la maggior parte degli sviluppatori si aspetta che un enum sia int.

Sintassi

- enum Colors {Red, Green, Blue} // Dichiarazione Enum
- enum Colori: byte {rosso, verde, blu} // Dichiarazione con tipo specifico
- enum Colors {Red = 23, Green = 45, Blue = 12} // Dichiarazione con valori definiti
- Colors.Red // Accede a un elemento di un Enum
- int value = (int) Colors.Red // Ottieni il valore int di un elemento enum
- Colors color = (Colors) intValue // Ottieni un elemento enum da int

Osservazioni

Un enum (abbreviazione di "tipo enumerato") è un tipo costituito da un insieme di costanti nominate, rappresentate da un identificatore specifico del tipo.

Le enumerazioni sono più utili per rappresentare concetti che hanno un numero (solitamente piccolo) di valori discreti possibili. Ad esempio, possono essere utilizzati per rappresentare un giorno della settimana o un mese dell'anno. Possono anche essere usati come flag che possono essere combinati o controllati, usando operazioni bit a bit.

Examples

Ottieni tutti i valori dei membri di un enum

```
enum MyEnum
{
    One,
    Two,
    Three
}

foreach(MyEnum e in Enum.GetValues(typeof(MyEnum)))
    Console.WriteLine(e);
```

Questo stamperà:

```
One
Two
Three
```

Enum come bandiere

Il `FlagsAttribute` può essere applicato a un enum che modifica il comportamento di `ToString()` in modo che corrisponda alla natura dell'enum:

```
[Flags]
enum MyEnum
{
    //None = 0, can be used but not combined in bitwise operations
    FlagA = 1,
    FlagB = 2,
    FlagC = 4,
    FlagD = 8
    //you must use powers of two or combinations of powers of two
    //for bitwise operations to work
}

var twoFlags = MyEnum.FlagA | MyEnum.FlagB;

// This will enumerate all the flags in the variable: "FlagA, FlagB".
Console.WriteLine(twoFlags);
```

Poiché `FlagsAttribute` si basa sulle costanti di enumerazione per essere poteri di due (o le loro combinazioni) ei valori enum sono in definitiva valori numerici, si è limitati dalla dimensione del tipo numerico sottostante. Il più grande tipo numerico disponibile che è possibile utilizzare è `UInt64` , che consente di specificare 64 costanti enum flag distinte (non combinate). La parola chiave `enum` imposta automaticamente il tipo sottostante `int` , che è `Int32` . Il compilatore consentirà la dichiarazione di valori più ampi di 32 bit. Quelli si avvolgeranno senza avviso e generano due o più membri enum dello stesso valore. Pertanto, se un enum è pensato per contenere un set di bit di oltre 32 flag, è necessario specificare esplicitamente un tipo più grande:

```
public enum BigEnum : ulong
{
    BigValue = 1 << 63
}
```

Sebbene i flag siano spesso solo un singolo bit, possono essere combinati in "set" denominati per un utilizzo più semplice.

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,
```

```
    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

Per evitare di compitare i valori decimali delle potenze di due, l' operatore di spostamento a sinistra (<<) può anche essere usato per dichiarare lo stesso enum

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1 << 0,
    Option2 = 1 << 1,
    Option3 = 1 << 2,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

A partire da C # 7.0, possono essere usati anche i letterali binari .

Per verificare se il valore della variabile enum ha un determinato flag impostato, è possibile utilizzare il metodo `HasFlag` . Diciamo che abbiamo

```
[Flags]
enum MyEnum
{
    One = 1,
    Two = 2,
    Three = 4
}
```

E un `value`

```
var value = MyEnum.One | MyEnum.Two;
```

Con `HasFlag` possiamo controllare se uno qualsiasi dei flag è impostato

```
if(value.HasFlag(MyEnum.One))
    Console.WriteLine("Enum has One");

if(value.HasFlag(MyEnum.Two))
    Console.WriteLine("Enum has Two");

if(value.HasFlag(MyEnum.Three))
    Console.WriteLine("Enum has Three");
```

Inoltre possiamo scorrere tutti i valori di enum per ottenere tutti i flag impostati

```
var type = typeof(MyEnum);
var names = Enum.GetNames(type);

foreach (var name in names)
{
```

```
    var item = (MyEnum)Enum.Parse(type, name);

    if (value.HasFlag(item))
        Console.WriteLine("Enum has " + name);
}
```

O

```
foreach(MyEnum flagToCheck in Enum.GetValues(typeof(MyEnum)))
{
    if(value.HasFlag(flagToCheck))
    {
         Console.WriteLine("Enum has " + flagToCheck);
    }
}
```

Tutti e tre gli esempi stamperanno:

```
Enum has One
Enum has Two
```

Prova i valori enum in stile flags con logica bit a bit

Un valore enum in stile flags deve essere testato con logica bit a bit perché potrebbe non corrispondere a nessun singolo valore.

```
[Flags]
enum FlagsEnum
{
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,
    Option2And3 = Option2 | Option3;

    Default = Option1 | Option3,
}
```

Il valore `Default` è in realtà una combinazione di altri due *uniti* con un OR bit a bit. Quindi per testare la presenza di un flag dobbiamo usare un AND bit a bit.

```
var value = FlagsEnum.Default;

bool isOption2And3Set = (value & FlagsEnum.Option2And3) == FlagsEnum.Option2And3;

Assert.True(isOption2And3Set);
```

Enum per corda e schiena

```
public enum DayOfWeek
{
    Sunday,
    Monday,
```

```
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}


// Enum to string
string thursday = DayOfWeek.Thursday.ToString(); // "Thursday"

string seventhDay = Enum.GetName(typeof(DayOfWeek), 6); // "Saturday"

string monday = Enum.GetName(typeof(DayOfWeek), DayOfWeek.Monday); // "Monday"


// String to enum (.NET 4.0+ only - see below for alternative syntax for earlier .NET
versions)
DayOfWeek tuesday;
Enum.TryParse("Tuesday", out tuesday); // DayOfWeek.Tuesday

DayOfWeek sunday;
bool matchFound1 = Enum.TryParse("SUNDAY", out sunday); // Returns false (case-sensitive
match)

DayOfWeek wednesday;
bool matchFound2 = Enum.TryParse("WEDNESDAY", true, out wednesday); // Returns true;
DayOfWeek.Wednesday (case-insensitive match)


// String to enum (all .NET versions)
DayOfWeek friday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Friday"); // DayOfWeek.Friday

DayOfWeek caturday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Caturady"); // Thows
ArgumentException

// All names of an enum type as strings
string[] weekdays = Enum.GetNames(typeof(DayOfWeek));
```

Valore predefinito per enum == ZERO

Il valore predefinito per un enum è zero . Se un enum non definisce un oggetto con un valore pari a zero, il suo valore predefinito sarà zero.

```
public class Program
{
    enum EnumExample
    {
        one = 1,
        two = 2
    }

    public void Main()
    {
        var e = default(EnumExample);

        if (e == EnumExample.one)
            Console.WriteLine("defaults to one");
        else
```

```
            Console.WriteLine("Unknown");
    }
}
```

Esempio: https://dotnetfiddle.net/l5Rwie

Nozioni di base su Enum

Da MSDN :

> Un tipo di enumerazione (chiamato anche enumerazione o enumerazione) fornisce un modo efficace per definire un insieme di **costanti integrali con** nome che possono essere **assegnate a una variabile** .

Essenzialmente, un enum è un tipo che consente solo un insieme di opzioni finite e ogni opzione corrisponde a un numero. Per impostazione predefinita, tali numeri aumentano nell'ordine in cui i valori sono dichiarati, a partire da zero. Ad esempio, si potrebbe dichiarare un enum per i giorni della settimana:

```
public enum Day
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

Quell'enum potrebbe essere usato così:

```
// Define variables with values corresponding to specific days
Day myFavoriteDay = Day.Friday;
Day myLeastFavoriteDay = Day.Monday;

// Get the int that corresponds to myFavoriteDay
// Friday is number 4
int myFavoriteDayIndex = (int)myFavoriteDay;

// Get the day that represents number 5
Day dayFive = (Day)5;
```

Per impostazione predefinita, il tipo sottostante di ciascun elemento `enum` è `int` , ma possono essere utilizzati anche `byte` , `sbyte` , `short` , `ushort` , `uint` , `long` e `ulong` . Se si utilizza un tipo diverso da `int` , è necessario specificare il tipo utilizzando i due punti dopo il nome dell'enumerazione:

```
public enum Day : byte
{
    // same as before
}
```

I numeri dopo il nome ora sono byte anziché interi. È possibile ottenere il tipo sottostante dell'enumerazione come segue:

```
Enum.GetUnderlyingType(typeof(Days)));
```

Produzione:

```
System.Byte
```

Demo: .NET violino

Manipolazione bit a bit tramite enumerazione

Il FlagsAttribute deve essere utilizzato ogni volta che l'enumerable rappresenta una raccolta di flag, piuttosto che un singolo valore. Il valore numerico assegnato a ciascun valore enumerato aiuta a manipolare le enumerazioni utilizzando operatori bit a bit.

Esempio 1: con [bandiere]

```
[Flags]
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

stampa rosso, blu

Esempio 2: senza [bandiere]

```
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}
var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

stampa 3

Usando la notazione << per le bandiere

L'operatore di spostamento a sinistra (`<<`) può essere utilizzato nelle dichiarazioni flag enum per garantire che ogni flag abbia esattamente un `1` nella rappresentazione binaria, come dovrebbero

essere le bandiere.

Questo aiuta anche a migliorare la leggibilità di enumerazioni di grandi dimensioni con un sacco di bandiere in esse.

```
[Flags]
public enum MyEnum
{
    None  = 0,
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2,
    Flag4 = 1 << 3,
    Flag5 = 1 << 4,
    ...
    Flag31 = 1 << 30
}
```

Ora è ovvio che `MyEnum` contiene solo flag appropriati e non roba disordinata come `Flag30 = 1073741822` (o 11111111111111111111111111111110 in binario) che è inappropriato.

Aggiunta di ulteriori informazioni di descrizione a un valore enum

In alcuni casi è possibile aggiungere una descrizione aggiuntiva a un valore enum, ad esempio quando il valore enum è meno leggibile di quello che si potrebbe desiderare di visualizzare all'utente. In questi casi è possibile utilizzare la classe `System.ComponentModel.DescriptionAttribute` .

Per esempio:

```
public enum PossibleResults
{
    [Description("Success")]
    OK = 1,
    [Description("File not found")]
    FileNotFound = 2,
    [Description("Access denied")]
    AccessDenied = 3
}
```

Ora, se vuoi restituire la descrizione di un valore enum specifico, puoi fare quanto segue:

```
public static string GetDescriptionAttribute(PossibleResults result)
{
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((result.GetType().GetField(result.ToString())),
typeof(DescriptionAttribute))).Description;
}

static void Main(string[] args)
{
    PossibleResults result = PossibleResults.FileNotFound;
    Console.WriteLine(result); // Prints "FileNotFound"
    Console.WriteLine(GetDescriptionAttribute(result)); // Prints "File not found"
}
```

Questo può anche essere facilmente trasformato in un metodo di estensione per tutte le enumerazioni:

```
static class EnumExtensions
{
    public static string GetDescription(this Enum enumValue)
    {
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((enumValue.GetType().GetField(enumValue.ToString())
typeof(DescriptionAttribute))).Description;
    }
}
```

E quindi facilmente utilizzabile in questo modo: `Console.WriteLine(result.GetDescription());`

Aggiungi e rimuovi i valori dall'enumerazione contrassegnata

Questo codice è per aggiungere e rimuovere un valore da un'istanza enum contrassegnata:

```
[Flags]
public enum MyEnum
{
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2
}

var value = MyEnum.Flag1;

// set additional value
value |= MyEnum.Flag2;  //value is now Flag1, Flag2
value |= MyEnum.Flag3;  //value is now Flag1, Flag2, Flag3

// remove flag
value &= ~MyEnum.Flag2; //value is now Flag1, Flag3
```

Le enumerazioni possono avere valori inaspettati

Poiché un enum può essere lanciato su e dal suo tipo integrale sottostante, il valore potrebbe non rientrare nell'intervallo di valori indicato nella definizione del tipo enum.

Sebbene il tipo di `DaysOfWeek` seguente `DaysOfWeek` abbia solo 7 valori definiti, può comunque contenere qualsiasi valore `int` .

```
public enum DaysOfWeek
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}
```

```
DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(d); // prints 31

DaysOFWeek s = DaysOfWeek.Sunday;
s++; // No error
```

Attualmente non esiste un modo per definire un enum che non abbia questo comportamento.

Tuttavia, i valori di enum non definiti possono essere rilevati utilizzando il metodo `Enum.IsDefined` . Per esempio,

```
DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(Enum.IsDefined(typeof(DaysOfWeek),d)); // prints False
```

Leggi enum online: https://riptutorial.com/it/csharp/topic/931/enum

Capitolo 46: Eredità

Sintassi

- class DerivedClass: BaseClass
- class DerivedClass: BaseClass, IExampleInterface
- class DerivedClass: BaseClass, IExampleInterface, IAnotherInterface

Osservazioni

Le classi possono ereditare direttamente da una sola classe, ma (al suo posto o allo stesso tempo) possono implementare una o più interfacce.

Le strutture possono implementare interfacce ma non possono ereditare esplicitamente da alcun tipo. Essi ereditano implicitamente da `System.ValueType` , che a sua volta eredita direttamente da `System.Object` .

Le classi statiche non possono implementare interfacce.

Examples

Ereditare da una classe base

Per evitare la duplicazione del codice, definire i metodi e gli attributi comuni in una classe generale come base:

```
public class Animal
{
    public string Name { get; set; }
    // Methods and attributes common to all animals
    public void Eat(Object dinner)
    {
        // ...
    }
    public void Stare()
    {
        // ...
    }
    public void Roll()
    {
        // ...
    }
}
```

Ora che hai una classe che rappresenta `Animal` in generale, puoi definire una classe che descrive le peculiarità di specifici animali:

```
public class Cat : Animal
```

```
{
    public Cat()
    {
        Name = "Cat";
    }
    // Methods for scratching furniture and ignoring owner
    public void Scratch(Object furniture)
    {
        // ...
    }
}
```

La classe Cat accede non solo ai metodi descritti nella sua definizione in modo esplicito, ma anche a tutti i metodi definiti nella classe generale `Animal` base. Qualsiasi animale (che fosse o meno un gatto) poteva mangiare, stare in piedi o rotolare. Un animale non sarebbe in grado di grattare, tuttavia, a meno che non fosse anche un gatto. Potresti quindi definire altre classi che descrivono altri animali. (Come Gopher con un metodo per distruggere i giardini fioriti e la bradipo senza alcun metodo aggiuntivo.)

Ereditare da una classe e implementare un'interfaccia

```
public class Animal
{
    public string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : Animal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

Ereditare da una classe e implementare più interfacce

```
public class LivingBeing
{
    string Name { get; set; }
}

public interface IAnimal
{
    bool HasHair { get; set; }
```

```
}

public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : LivingBeing, IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
        HasHair = true;
    }

    public bool HasHair { get; set; }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

Test e navigazione dell'eredità

```
interface BaseInterface {}
class BaseClass : BaseInterface {}

interface DerivedInterface {}
class DerivedClass : BaseClass, DerivedInterface {}

var baseInterfaceType = typeof(BaseInterface);
var derivedInterfaceType = typeof(DerivedInterface);
var baseType = typeof(BaseClass);
var derivedType = typeof(DerivedClass);

var baseInstance = new BaseClass();
var derivedInstance = new DerivedClass();

Console.WriteLine(derivedInstance is DerivedClass); //True
Console.WriteLine(derivedInstance is DerivedInterface); //True
Console.WriteLine(derivedInstance is BaseClass); //True
Console.WriteLine(derivedInstance is BaseInterface); //True
Console.WriteLine(derivedInstance is object); //True

Console.WriteLine(derivedType.BaseType.Name);  //BaseClass
Console.WriteLine(baseType.BaseType.Name);  //Object
Console.WriteLine(typeof(object).BaseType);  //null

Console.WriteLine(baseType.IsInstanceOfType(derivedInstance));  //True
Console.WriteLine(derivedType.IsInstanceOfType(baseInstance));  //False

Console.WriteLine(
    string.Join(",",
    derivedType.GetInterfaces().Select(t => t.Name).ToArray()));
//BaseInterface,DerivedInterface
```

```
Console.WriteLine(baseInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(baseType)); //False
```

Estendere una classe base astratta

A differenza delle interfacce, che possono essere descritte come contratti per l'implementazione, le classi astratte fungono da contratti per l'estensione.

Una classe astratta non può essere istanziata, deve essere estesa e la classe risultante (o classe derivata) può quindi essere istanziata.

Le classi astratte vengono utilizzate per fornire implementazioni generiche

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }
}

public class Mustang : Car
{
    // Simply by extending the abstract class Car, the Mustang can HonkHorn()
    // If Car were an interface, the HonkHorn method would need to be included
    // in every class that implemented it.
}
```

L'esempio sopra mostra come qualsiasi classe che estende l'auto riceverà automaticamente il metodo HonkHorn con l'implementazione. Ciò significa che qualsiasi sviluppatore che crea una nuova auto non dovrà preoccuparsi di come suonerà il clacson.

Costruttori in una sottoclasse

Quando crei una sottoclasse di una classe base, puoi costruire la classe base usando `: base` dopo i parametri del costruttore della sottoclasse.

```
class Instrument
{
    string type;
    bool clean;

    public Instrument (string type, bool clean)
    {
        this.type = type;
        this.clean = clean;
    }
}

class Trumpet : Instrument
{
    bool oiled;
```

```
    public Trumpet(string type, bool clean, bool oiled) : base(type, clean)
    {
        this.oiled = oiled;
    }
}
```

Eredità. Sequenza di chiamate dei costruttori

Considera di avere un `Animal` classe che ha un `Dog` classe

```
class Animal
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}

class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("In Dog's constructor");
    }
}
```

Per impostazione predefinita, ogni classe eredita implicitamente la classe `Object` .

Questo è uguale al codice precedente.

```
class Animal : Object
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}
```

Quando si crea un'istanza della classe `Dog` , il **costruttore predefinito delle classi base (senza parametri) verrà chiamato se non vi è alcuna chiamata esplicita a un altro costruttore nella classe genitore** . Nel nostro caso, prima si chiamerà `Object's` constructor, quindi `Animal's` e alla fine `Dog's` costruttore `Dog's` .

```
public class Program
{
    public static void Main()
    {
        Dog dog = new Dog();
    }
}
```

L'output sarà

> Nel costruttore di Animal

Nel costruttore di Dog

Visualizza la demo

Chiama il costruttore genitore in modo esplicito.

Negli esempi precedenti, il nostro costruttore di classi `Dog` chiama il costruttore **predefinito** della classe `Animal` . Se vuoi, puoi specificare quale costruttore dovrebbe essere chiamato: è possibile chiamare qualsiasi costruttore che è definito nella classe genitore.

Considera che abbiamo queste due classi.

```
class Animal
{
    protected string name;

    public Animal()
    {
        Console.WriteLine("Animal's default constructor");
    }

    public Animal(string name)
    {
        this.name = name;
        Console.WriteLine("Animal's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}

class Dog : Animal
{
    public Dog() : base()
    {
        Console.WriteLine("Dog's default constructor");
    }

    public Dog(string name) : base(name)
    {
        Console.WriteLine("Dog's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}
```

Cosa sta succedendo qui?

Abbiamo 2 costruttori in ogni classe.

Cosa significa `base` ?

`base` è un riferimento alla classe genitore. Nel nostro caso, quando creiamo un'istanza di classe `Dog` come questa

```
Dog dog = new Dog();
```

Il runtime chiama prima `Dog()` , che è il costruttore senza parametri. Ma il suo corpo non funziona

immediatamente. Dopo le parentesi del costruttore abbiamo una tale chiamata: `base()` , il che significa che quando chiamiamo il costruttore predefinito di `Dog` , a sua volta chiamerà il costruttore **predefinito** del genitore. Dopo che il costruttore del genitore è stato eseguito, verrà restituito e, infine, verrà eseguito il corpo del costruttore `Dog()` .

Quindi l'output sarà così:

> Costruttore predefinito di Animal
> Costruttore predefinito del cane

Visualizza la demo

E se chiamassimo il costruttore `Dog's` con un parametro?

```
Dog dog = new Dog("Rex");
```

Sapete che i membri della classe genitore che non sono privati sono ereditati dalla classe figlia, nel senso che `Dog` avrà anche il campo del `name` .
In questo caso abbiamo passato un argomento al nostro costruttore. A sua volta passa l'argomento al **costruttore della** classe genitore **con un parametro** , che inizializza il campo del `name` .

L'output sarà

```
Animal's constructor with 1 parameter
Rex
Dog's constructor with 1 parameter
Rex
```

Sommario:

Ogni creazione di oggetti inizia dalla classe base. Nell'ereditarietà, le classi che si trovano nella gerarchia sono concatenate. Poiché tutte le classi derivano da `Object` , il primo costruttore da chiamare quando viene creato un oggetto è il costruttore della classe `Object` ; Quindi viene chiamato il prossimo costruttore della catena e solo dopo che tutti sono chiamati viene creato l'oggetto

parola chiave di base

1. La parola chiave di base viene utilizzata per accedere ai membri della classe base da una classe derivata:
2. Chiama un metodo sulla classe base che è stato sovrascritto da un altro metodo. Specificare quale costruttore della classe base deve essere chiamato quando si creano le istanze della classe derivata.

Metodi ereditari

Esistono diversi modi in cui i metodi possono essere ereditati

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }

    // virtual methods CAN be overridden in derived classes
    public virtual void ChangeGear() {
        // Implementation of gears being changed
    }

    // abstract methods MUST be overridden in derived classes
    public abstract void Accelerate();
}

public class Mustang : Car
{
    // Before any code is added to the Mustang class, it already contains
    // implementations of HonkHorn and ChangeGear.

    // In order to compile, it must be given an implementation of Accelerate,
    // this is done using the override keyword
    public override void Accelerate() {
        // Implementation of Mustang accelerating
    }

    // If the Mustang changes gears differently to the implementation in Car
    // this can be overridden using the same override keyword as above
    public override void ChangeGear() {
        // Implementation of Mustang changing gears
    }
}
```

Ereditarietà Anti-pattern

Eredità impropria

Diciamo che ci sono 2 classi di classe `Foo` e `Bar` . `Foo` ha due funzioni `Do1` e `Do2` . `Bar` bisogno di usare `Do1` da `Foo` , ma non ha bisogno di `Do2` o di funzionalità che equivalgono a `Do2` ma fa qualcosa di completamente diverso.

Cattivo : crea `Do2()` su `Foo` virtual quindi sostituiscilo in `Bar` o `throw Exception` in `Bar` per `Do2()`

```
public class Bar : Foo
{
    public override void Do2()
    {
        //Does something completely different that you would expect Foo to do
        //or simply throws new Exception
    }
}
```

Buon modo

Prendi `Do1()` da `Foo` e mettilo nella nuova classe `Baz` poi eredita sia `Foo` che `Bar` da `Baz` e implementa

`Do2()` separatamente

```
public class Baz
{
    public void Do1()
    {
        // magic
    }
}

public class Foo : Baz
{
    public void Do2()
    {
        // foo way
    }
}

public class Bar : Baz
{
    public void Do2()
    {
        // bar way or not have Do2 at all
    }
}
```

Ora, perché il primo esempio è cattivo e il secondo è buono: quando lo sviluppatore nr2 deve fare una modifica in `Foo` , è probabile che interromperà l'implementazione di `Bar` perché la `Bar` è ora inseparabile da `Foo` . Quando lo si fa con l'ultimo esempio, `Foo` and `Bar` commonalty è stato spostato su `Baz` e non si influenzano a vicenda (come il non dovrebbe).

Classe base con specifica del tipo ricorsivo

Definizione una tantum di una classe base generica con identificatore di tipo ricorsivo. Ogni nodo ha un genitore e più figli.

```
/// <summary>
/// Generic base class for a tree structure
/// </summary>
/// <typeparam name="T">The node type of the tree</typeparam>
public abstract class Tree<T> where T : Tree<T>
{
    /// <summary>
    /// Constructor sets the parent node and adds this node to the parent's child nodes
    /// </summary>
    /// <param name="parent">The parent node or null if a root</param>
    protected Tree(T parent)
    {
        this.Parent=parent;
        this.Children=new List<T>();
        if(parent!=null)
        {
            parent.Children.Add(this as T);
        }
    }
    public T Parent { get; private set; }
    public List<T> Children { get; private set; }
```

```
    public bool IsRoot { get { return Parent==null; } }
    public bool IsLeaf { get { return Children.Count==0; } }
    /// <summary>
    /// Returns the number of hops to the root object
    /// </summary>
    public int Level { get { return IsRoot ? 0 : Parent.Level+1; } }
}
```

Quanto sopra può essere riutilizzato ogni volta che è necessario definire una gerarchia di alberi. L'oggetto nodo nella struttura deve ereditare dalla classe base con

```
public class MyNode : Tree<MyNode>
{
    // stuff
}
```

ogni classe nodo sa dove si trova nella gerarchia, quale sia l'oggetto genitore e gli oggetti figli. Diversi tipi incorporati utilizzano una struttura ad albero, come `Control` o `XmlElement` e l' `Tree<T>` sopra può essere usato come una classe base di *qualsiasi* tipo nel codice.

Ad esempio, per creare una gerarchia di parti in cui viene calcolato il peso totale dal peso di *tutti* i bambini, effettuare le seguenti operazioni:

```
public class Part : Tree<Part>
{
    public static readonly Part Empty = new Part(null) { Weight=0 };
    public Part(Part parent) : base(parent) { }
    public Part Add(float weight)
    {
        return new Part(this) { Weight=weight };
    }
    public float Weight { get; set; }

    public float TotalWeight { get { return Weight+Children.Sum((part) => part.TotalWeight); }
}
}
```

essere usato come

```
// [Q:2.5] -- [P:4.2] -- [R:0.4]
//    \
//      - [Z:0.8]
var Q = Part.Empty.Add(2.5f);
var P = Q.Add(4.2f);
var R = P.Add(0.4f);
var Z = Q.Add(0.9f);

// 2.5+(4.2+0.4)+0.9 = 8.0
float weight = Q.TotalWeight;
```

Un altro esempio sarebbe nella definizione dei frame di coordinate relative. In questo caso, la posizione reale del frame di coordinate dipende dalle posizioni di *tutti* i frame di coordinate padre.

```
public class RelativeCoordinate : Tree<RelativeCoordinate>
{
    public static readonly RelativeCoordinate Start = new RelativeCoordinate(null,
PointF.Empty) { };
    public RelativeCoordinate(RelativeCoordinate parent, PointF local_position)
        : base(parent)
    {
        this.LocalPosition=local_position;
    }
    public PointF LocalPosition { get; set; }
    public PointF GlobalPosition
    {
        get
        {
            if(IsRoot) return LocalPosition;
            var parent_pos = Parent.GlobalPosition;
            return new PointF(parent_pos.X+LocalPosition.X, parent_pos.Y+LocalPosition.Y);
        }
    }
    public float TotalDistance
    {
        get
        {
            float dist =
(float)Math.Sqrt(LocalPosition.X*LocalPosition.X+LocalPosition.Y*LocalPosition.Y);
            return IsRoot ? dist : Parent.TotalDistance+dist;
        }
    }
    public RelativeCoordinate Add(PointF local_position)
    {
        return new RelativeCoordinate(this, local_position);
    }
    public RelativeCoordinate Add(float x, float y)
    {
        return Add(new PointF(x, y));
    }
}
```

essere usato come

```
// Define the following coordinate system hierarchy
//
// o--> [A1] --+--> [B1] -----> [C1]
//             |
//             +--> [B2] --+--> [C2]
//                         |
//                         +--> [C3]

var A1 = RelativeCoordinate.Start;
var B1 = A1.Add(100, 20);
var B2 = A1.Add(160, 10);

var C1 = B1.Add(120, -40);
var C2 = B2.Add(80, -20);
var C3 = B2.Add(60, -30);

double dist1 = C1.TotalDistance;
```

Leggi Eredità online: https://riptutorial.com/it/csharp/topic/29/eredita

Capitolo 47: Esecuzione di richieste HTTP

Examples

Creazione e invio di una richiesta POST HTTP

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/submit.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);
request.Method = "POST";

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.Deflate | DecompressionMethods.GZip;
request.ContentType = "application/x-www-form-urlencoded";
// Could also set other HTTP headers such as Request.UserAgent, Request.Referer,
// Request.Accept, or other headers via the Request.Headers collection.

// Set the POST request body data. In this example, the POST data is in
// application/x-www-form-urlencoded format.
string postData = "myparam1=myvalue1&myparam2=myvalue2";
using (var writer = new StreamWriter(request.GetRequestStream()))
{
    writer.Write(postData);
}

// Submit the request, and get the response body from the remote server.
string responseFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseFromRemoteServer = reader.ReadToEnd();
    }
}
```

Creazione e invio di una richiesta GET HTTP

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/page.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
request.Timeout = 2 * 60 * 1000; // 2 minutes, in milliseconds

// Submit the request, and get the response body.
string responseBodyFromRemoteServer;
```

```
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseBodyFromRemoteServer = reader.ReadToEnd();
    }
}
```

Gestione degli errori di specifici codici di risposta HTTP (come 404 non trovato)

```
using System.Net;

...

string serverResponse;
try
{
    // Call a method that performs an HTTP request (per the above examples).
    serverResponse = PerformHttpRequest();
}
catch (WebException ex)
{
    if (ex.Status == WebExceptionStatus.ProtocolError)
    {
        HttpWebResponse response = ex.Response as HttpWebResponse;
        if (response != null)
        {
            if ((int)response.StatusCode == 404) // Not Found
            {
                // Handle the 404 Not Found error
                // ...
            }
            else
            {
                // Could handle other response.StatusCode values here.
                // ...
            }
        }
    }
    else
    {
        // Could handle other error conditions here, such as
WebExceptionStatus.ConnectFailure.
        // ...
    }
}
```

Invio di richieste POST HTTP asincrone con corpo JSON

```
public static async Task PostAsync(this Uri uri, object value)
{
    var content = new ObjectContext(value.GetType(), value, new JsonMediaTypeFormatter());

    using (var client = new HttpClient())
    {
        return await client.PostAsync(uri, content);
```

```
    }
}

. . .

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-
requests");
await uri.PostAsync(new { foo = 123.45, bar = "Richard Feynman" });
```

Invio di richiesta HTTP GET asincrona e lettura della richiesta JSON

```
public static async Task<TResult> GetAnsync<TResult>(this Uri uri)
{
    using (var client = new HttpClient())
    {
        var message = await client.GetAsync(uri);

        if (!message.IsSuccessStatusCode)
            throw new Exception();

        return message.ReadAsAsync<TResult>();
    }
}

. . .

public class Result
{
    public double foo { get; set; }

    public string bar { get; set; }
}

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-
requests");
var result = await uri.GetAsync<Result>();
```

Recupera HTML per pagina Web (semplice)

```
string contents = "";
string url = "http://msdn.microsoft.com";

using (System.Net.WebClient client = new System.Net.WebClient())
{
    contents = client.DownloadString(url);
}

Console.WriteLine(contents);
```

Leggi Esecuzione di richieste HTTP online: https://riptutorial.com/it/csharp/topic/1971/esecuzione-di-richieste-http

Capitolo 48: Esempi Async / Waitit, Backgroundworker, Task e Thread

Osservazioni

Per eseguire uno di questi esempi, chiamali in questo modo:

```
static void Main()
{
    new Program().ProcessDataAsync();
    Console.ReadLine();
}
```

Examples

ASP.NET Configure Await

Quando ASP.NET gestisce una richiesta, viene assegnato un thread dal pool di thread e viene creato un **contesto di richiesta** . Il contesto della richiesta contiene informazioni sulla richiesta corrente a cui è possibile accedere tramite la proprietà statica `HttpContext.Current` . Il contesto della richiesta per la richiesta viene quindi assegnato al thread che gestisce la richiesta.

Un determinato contesto di richiesta **può essere attivo solo su un thread alla volta** .

Quando l'esecuzione giunge in `await` , il thread che gestisce una richiesta viene restituito al pool di thread mentre viene eseguito il metodo asincrono e il contesto della richiesta è gratuito per un altro thread da utilizzare.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    var products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    // Execution continues using the original request context.
    return View(products);
}
```

Al termine dell'attività, il pool di thread assegna un altro thread per continuare l'esecuzione della richiesta. Il contesto della richiesta viene quindi assegnato a questo thread. Questo potrebbe essere o non essere il thread originale.

Blocco

Quando si attende il risultato di una chiamata al metodo `async` , possono verificarsi deadlock **sincroni** . Ad esempio, il seguente codice genererà un deadlock quando viene chiamato

`IndexSync()` :

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

Questo perché, per impostazione predefinita, l'attività attesa, in questo caso `db.Products.ToListAsync()` acquisisce il contesto (nel caso di ASP.NET il contesto della richiesta) e tenta di utilizzarlo una volta completato.

Quando l'intero stack di chiamate è asincrono, non ci sono problemi perché, una volta in `await` viene raggiunto, il thread originale viene rilasciato, liberando il contesto della richiesta.

Quando si blocca in modo sincrono utilizzando `Task.Result` o `Task.Wait()` (o altri metodi di blocco) il thread originale è ancora attivo e conserva il contesto della richiesta. Il metodo atteso funziona ancora in modo asincrono e una volta che il callback tenta di essere eseguito, ovvero quando viene restituita l'attività attesa, tenta di ottenere il contesto della richiesta.

Pertanto il deadlock si verifica perché mentre il thread di blocco con il contesto della richiesta è in attesa del completamento dell'operazione asincrona, l'operazione asincrona sta tentando di ottenere il contesto della richiesta per il completamento.

ConfigureAwait

Per impostazione predefinita, le chiamate a un'attività attesa acquisiscono il contesto corrente e tentano di riprendere l'esecuzione nel contesto una volta completato.

Usando `ConfigureAwait(false)` questo può essere prevenuto e le deadlock possono essere evitate.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync().ConfigureAwait(false);

    // Execution resumes on a "random" thread from the pool without the original request
context
    return View(products);
}
```

```
public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

Questo può evitare deadlock quando è necessario bloccare il codice asincrono, tuttavia ciò comporta il costo di perdere il contesto nella continuazione (codice dopo la chiamata in attesa).

In ASP.NET questo significa che se il tuo codice segue una chiamata per `await someTask.ConfigureAwait(false);` tenta di accedere alle informazioni dal contesto, ad esempio `HttpContext.Current.User` quindi le informazioni sono state perse. In questo caso, `HttpContext.Current` è null. Per esempio:

```
public async Task<ActionResult> Index()
{
    // Contains information about the user sending the request
    var user = System.Web.HttpContext.Current.User;

    using (var client = new HttpClient())
    {
        await client.GetAsync("http://google.com").ConfigureAwait(false);
    }

    // Null Reference Exception, Current is null
    var user2 = System.Web.HttpContext.Current.User;

    return View();
}
```

Se viene utilizzato `ConfigureAwait(true)` (equivalente a non avere affatto ConfigureAwait), sia l' `user` che l' `user2` vengono popolati con gli stessi dati.

Per questo motivo, si consiglia spesso di utilizzare `ConfigureAwait(false)` nel codice della libreria in cui il contesto non è più utilizzato.

Async / await

Vedi sotto per un semplice esempio di come usare async / attendi di fare un po 'di tempo in un processo in background, mantenendo l'opzione di fare altre cose che non hanno bisogno di aspettare le cose che richiedono molto tempo per essere completate.

Tuttavia, se è necessario lavorare con il risultato del metodo intensivo in un secondo momento, è possibile farlo attendendo l'esecuzione.

```
public async Task ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> task = TimeintensiveMethod(@"PATH_TO_SOME_FILE");
```

```
    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");

    // Wait for TimeintensiveMethod to complete and get its result
    int x = await task;
    Console.WriteLine("Count: " + x);
}

private async Task<int> TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = await reader.ReadToEndAsync();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

BackgroundWorker

Vedi sotto per un semplice esempio di come utilizzare un oggetto `BackgroundWorker` per eseguire operazioni che richiedono molto tempo in un thread in background.

Devi:

1. Definire un metodo di lavoro che `DoWork` il lavoro intensivo e chiamarlo da un gestore di eventi per l'evento `DoWork` di un `BackgroundWorker` .
2. Inizia l'esecuzione con `RunWorkerAsync` . Qualsiasi argomento richiesto dal metodo lavoratore attaccato `DoWork` possono essere trasmesse tramite la `DoWorkEventArgs` parametro `RunWorkerAsync` .

Oltre all'evento `DoWork` , la classe `BackgroundWorker` definisce anche due eventi che dovrebbero essere utilizzati per interagire con l'interfaccia utente. Questi sono opzionali.

- L'evento `RunWorkerCompleted` viene attivato quando i gestori `DoWork` sono stati completati.
- L'evento `ProgressChanged` viene attivato quando viene chiamato il metodo `ReportProgress` .

```
public void ProcessDataAsync()
{
    // Start the time intensive method
    BackgroundWorker bw = new BackgroundWorker();
    bw.DoWork += BwDoWork;
    bw.RunWorkerCompleted += BwRunWorkerCompleted;
    bw.RunWorkerAsync(@"PATH_TO_SOME_FILE");

    // Control returns here before TimeintensiveMethod returns
```

```
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

// Method that will be called after BwDoWork exits
private void BwRunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    // we can access possible return values of our Method via the Parameter e
    Console.WriteLine("Count: " + e.Result);
}

// execution of our time intensive Method
private void BwDoWork(object sender, DoWorkEventArgs e)
{
    e.Result = TimeintensiveMethod(e.Argument);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

       for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

Compito

Vedi sotto per un semplice esempio di come usare un `Task` per fare cose che richiedono molto tempo in un processo in background.

Tutto quello che devi fare è avvolgere il tuo metodo intensivo in una chiamata `Task.Run()` .

```
public void ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> t = Task.Run(() => TimeintensiveMethod(@"PATH_TO_SOME_FILE"));

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");

    Console.WriteLine("Count: " + t.Result);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
```

```
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

Filo

Vedi sotto per un semplice esempio di come usare un `Thread` per fare cose che richiedono tempo in un processo in background.

```
public async void ProcessDataAsync()
{
    // Start the time intensive method
    Thread t = new Thread(TimeintensiveMethod);

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

private void TimeintensiveMethod()
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(@"PATH_TO_SOME_FILE"))
    {
        string v = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            v.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");
}
```

Come puoi vedere, non possiamo restituire un valore dal nostro `TimeIntensiveMethod` perché `Thread` aspetta un metodo void come parametro.

Per ottenere un valore di ritorno da una `Thread` utilizzare un evento o il seguente:

```
int ret;
Thread t= new Thread(() =>
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
```

```
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something to demonstrate the coolness of await-async
    ret = new Random().Next(100);
});

t.Start();
t.Join(1000);
Console.Writeline("Count: " + ret);
```

Attività "esegui e dimentica" l'estensione

In alcuni casi (ad esempio la registrazione) potrebbe essere utile eseguire l'attività e non attendere il risultato. La seguente estensione consente di eseguire attività e continuare l'esecuzione del codice di resto:

```
public static class TaskExtensions
{
    public static async void RunAndForget(
        this Task task, Action<Exception> onException = null)
    {
        try
        {
            await task;
        }
        catch (Exception ex)
        {
            onException?.Invoke(ex);
        }
    }
}
```

Il risultato è atteso solo all'interno del metodo di estensione. Poiché `async` / `await` viene utilizzato, è possibile rilevare un'eccezione e chiamare un metodo opzionale per gestirlo.

Un esempio di come utilizzare l'estensione:

```
var task = Task.FromResult(0); // Or any other task from e.g. external lib.
task.RunAndForget(
    e =>
    {
        // Something went wrong, handle it.
    });
```

Leggi Esempi Async / Waitit, Backgroundworker, Task e Thread online:
https://riptutorial.com/it/csharp/topic/3824/esempi-async---waitit--backgroundworker--task-e-thread

Capitolo 49: Espressioni Lambda

Osservazioni

Un'espressione lambda è una sintassi per la creazione di funzioni anonime in linea. Più formalmente, dalla Guida alla Programmazione C # :

> Un'espressione lambda è una funzione anonima che è possibile utilizzare per creare delegati o tipi di albero di espressioni. Utilizzando espressioni lambda, è possibile scrivere funzioni locali che possono essere passate come argomenti o restituite come valore delle chiamate di funzione.

Un'espressione lambda viene creata usando l'operatore `=>` . Metti tutti i parametri sul lato sinistro dell'operatore. Sul lato destro, metti un'espressione che può usare quei parametri; questa espressione si risolverà come valore di ritorno della funzione. Più raramente, se necessario, un intero `{code block}` può essere utilizzato sul lato destro. Se il tipo di reso non è nullo, il blocco conterrà un'istruzione di reso.

Examples

Passare un'espressione lambda come parametro di un metodo

```
List<int> l2 = l1.FindAll(x => x > 6);
```

Qui `x => x > 6` è un'espressione lambda che funge da predicato che assicura che vengano restituiti solo gli elementi superiori a 6.

Lambda Expressions come abbreviazione per l'inizializzazione dei delegati

```
public delegate int ModifyInt(int input);
ModifyInt multiplyByTwo = x => x * 2;
```

La sintassi di espressione Lambda di cui sopra è equivalente al seguente codice verboso:

```
public delegate int ModifyInt(int input);

ModifyInt multiplyByTwo = delegate(int x){
    return x * 2;
};
```

Lambda sia per `Func` che per` Action`

Tipicamente i lambda sono usati per definire semplici *funzioni* (generalmente nel contesto di un'espressione linq):

```
var incremented = myEnumerable.Select(x => x + 1);
```

Qui il `return` è implicito.

Tuttavia, è anche possibile passare *azioni* come lambda:

```
myObservable.Do(x => Console.WriteLine(x));
```

Espressioni Lambda con parametri multipli o nessun parametro

Utilizza le parentesi intorno all'espressione a sinistra dell'operatore `=>` per indicare più parametri.

```
delegate int ModifyInt(int input1, int input2);
ModifyInt multiplyTwoInts = (x,y) => x * y;
```

Allo stesso modo, un insieme vuoto di parentesi indica che la funzione non accetta parametri.

```
delegate string ReturnString();
ReturnString getGreeting = () => "Hello world.";
```

Metti più dichiarazioni in una dichiarazione Lambda

A differenza di un'espressione lambda, una dichiarazione lambda può contenere più istruzioni separate da punto e virgola.

```
delegate void ModifyInt(int input);

ModifyInt addOneAndTellMe = x =>
{
    int result = x + 1;
    Console.WriteLine(result);
};
```

Si noti che le istruzioni sono racchiuse tra parentesi `{}` .

Ricorda che l'istruzione lambda non può essere utilizzata per creare alberi di espressione.

Lambdas può essere emesso sia come `Func` che come` Expression`

Supponendo la seguente classe `Person` :

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Il seguente lambda:

```
p => p.Age > 18
```

Può essere passato come argomento per entrambi i metodi:

```
public void AsFunc(Func<Person, bool> func)
public void AsExpression(Expression<Func<Person, bool>> expr)
```

Perché il compilatore è in grado di trasformare lambda sia in delegati che in `Expression` .

Ovviamente, i provider LINQ fanno molto affidamento su `Expression` s (esposti principalmente attraverso l'interfaccia `IQueryable<T>`) al fine di essere in grado di analizzare le query e tradurle per memorizzare le query.

Espressione Lambda come gestore di eventi

Le espressioni Lambda possono essere utilizzate per gestire eventi, il che è utile quando:

- Il gestore è corto.
- Il gestore non deve mai essere annullato.

Di seguito viene fornita una buona situazione in cui è possibile utilizzare un gestore di eventi lambda:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
```

Se è necessario annullare la sottoscrizione di un gestore di eventi registrato in un punto futuro del codice, l'espressione del gestore di eventi deve essere salvata in una variabile e la registrazione / annullamento della registrazione eseguita tramite tale variabile:

```
EventHandler handler = (sender, args) => Console.WriteLine("Email sent");

smtpClient.SendCompleted += handler;
smtpClient.SendCompleted -= handler;
```

La ragione per cui questo è fatto piuttosto che semplicemente ridigitare l'espressione lambda letteralmente per cancellarla (`-=`) è che il compilatore C # non considererà necessariamente le due espressioni uguali:

```
EventHandler handlerA = (sender, args) => Console.WriteLine("Email sent");
EventHandler handlerB = (sender, args) => Console.WriteLine("Email sent");
Console.WriteLine(handlerA.Equals(handlerB)); // May return "False"
```

Si noti che se vengono aggiunte istruzioni addizionali all'espressione lambda, le parentesi graffe circostanti richieste possono essere omesse accidentalmente, senza causare errori in fase di compilazione. Per esempio:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
emailSendButton.Enabled = true;
```

Questo verrà compilato, ma comporterà l'aggiunta dell'espressione lambda `(sender, args) => Console.WriteLine("Email sent");` come gestore di eventi ed esecuzione della dichiarazione

`emailSendButton.Enabled = true;` subito. Per risolvere questo problema, il contenuto del lambda deve essere racchiuso tra parentesi graffe. Questo può essere evitato usando le parentesi graffe dall'inizio, facendo attenzione quando si aggiungono ulteriori istruzioni a un gestore di eventi lambda o circondando il lambda in parentesi tonde dall'inizio:

```
smtpClient.SendCompleted += ((sender, args) => Console.WriteLine("Email sent"));
//Adding an extra statement will result in a compile-time error
```

Leggi Espressioni Lambda online: https://riptutorial.com/it/csharp/topic/46/espressioni-lambda

Capitolo 50: eventi

introduzione

Un evento è una notifica che qualcosa si è verificato (come un clic del mouse) o, in alcuni casi, sta per verificarsi (come una variazione di prezzo).

Le classi possono definire eventi e le loro istanze (oggetti) possono generare questi eventi. Ad esempio, un pulsante può contenere un evento Click che viene generato quando un utente ha fatto clic su di esso.

I gestori di eventi sono quindi i metodi che vengono richiamati quando viene generato l'evento corrispondente. Un modulo può contenere un gestore di eventi Clicked per ogni pulsante che contiene, per esempio.

Parametri

Parametro	Dettagli
EventArgsT	Il tipo che deriva da EventArgs e contiene i parametri dell'evento.
Nome dell'evento	Il nome dell'evento.
handlerName	Il nome del gestore di eventi.
SenderObject	L'oggetto che sta invocando l'evento.
EventArguments	Un'istanza del tipo EventArgsT che contiene i parametri dell'evento.

Osservazioni

Quando si alza un evento:

- Controllare sempre se il delegato è `null` . Un delegato nullo significa che l'evento non ha sottoscrittori. Aumentare un evento senza abbonati comporterà una `NullReferenceException` .

6.0

- Copia il delegato (ad es. `EventName`) in una variabile locale (ad es. `eventName`) prima di verificare la presenza di null / raise dell'evento. Questo evita condizioni di gara in ambienti multi-thread:

Sbagliato :

```
    if(Changed != null)      // Changed has 1 subscriber at this point
                             // In another thread, that one subscriber decided to unsubscribe
```

```
        Changed(this, args); // `Changed` is now null, `NullReferenceException` is thrown.
```

A destra :

```
    // Cache the "Changed" event as a local. If it is not null, then use
    // the LOCAL variable (handler) to raise the event, NOT the event itself.
    var handler = Changed;
    if(handler != null)
        handler(this, args);
```

6.0

- Utilizzare l'operatore null-condizionale (?.) Per aumentare il metodo anziché il controllo null del delegato per i sottoscrittori in un'istruzione `if` : `EventName?.Invoke(SenderObject, new EventArgsT());`
- Quando si utilizza l'azione <> per dichiarare i tipi di delegato, la firma del gestore di un metodo / evento anonimo deve essere uguale al tipo di delegato anonimo dichiarato nella dichiarazione di evento.

Examples

Dichiarare e sollevare eventi

Dichiarazione di un evento

Puoi dichiarare un evento su qualsiasi `class` o `struct` usando la seguente sintassi:

```
public class MyClass
{
    // Declares the event for MyClass
    public event EventHandler MyEvent;

    // Raises the MyEvent event
    public void RaiseEvent()
    {
        OnMyEvent();
    }
}
```

Esiste una sintassi espansa per la dichiarazione degli eventi, in cui si detiene un'istanza privata dell'evento e si definisce un'istanza pubblica utilizzando `add` e `set` accessors. La sintassi è molto simile alle proprietà C #. In tutti i casi, la sintassi sopra illustrata dovrebbe essere preferita, poiché il compilatore emette il codice per garantire che più thread possano aggiungere e rimuovere in modo sicuro i gestori di eventi all'evento della classe.

Alzare l'evento

6.0

```
private void OnMyEvent()
{
    EventName?.Invoke(this, EventArgs.Empty);
}
```

6.0

```
private void OnMyEvent()
{
    // Use a local for EventName, because another thread can modify the
    // public EventName between when we check it for null, and when we
    // raise the event.
    var eventName = EventName;

    // If eventName == null, then it means there are no event-subscribers,
    // and therefore, we cannot raise the event.
    if(eventName != null)
        eventName(this, EventArgs.Empty);

}
```

Tieni presente che gli eventi possono essere generati solo dal tipo di dichiarazione. I clienti possono solo iscriversi / annullare l'iscrizione.

Per le versioni C # precedenti alla 6.0, dove `EventName?.Invoke` non è supportato, è buona pratica assegnare l'evento a una variabile temporanea prima del richiamo, come mostrato nell'esempio, che garantisce la sicurezza del thread nei casi in cui più thread eseguono lo stesso codice. In caso `NullReferenceException` possibile che venga generata una `NullReferenceException` in alcuni casi in cui più thread utilizzano la stessa istanza di oggetto. In C # 6.0, il compilatore emette un codice simile a quello mostrato nell'esempio di codice per C # 6.

Dichiarazione di eventi standard

Dichiarazione di evento:

```
public event EventHandler<EventArgsT> EventName;
```

Dichiarazione del gestore eventi:

```
public void HandlerName(object sender, EventArgsT args) { /* Handler logic */ }
```

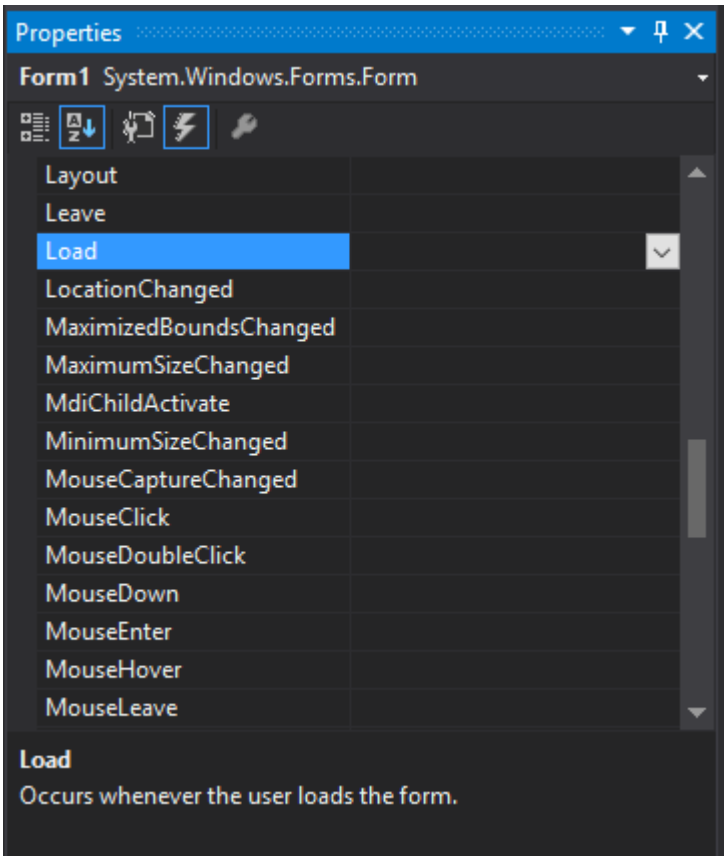
Iscrizione all'evento:

dinamicamente:

```
EventName += HandlerName;
```

Attraverso il Designer:

1. Fare clic sul pulsante Eventi nella finestra delle proprietà del controllo (Lightning bolt)
2. Fai doppio clic sul nome dell'evento:



3. Visual Studio genererà il codice evento:

```
private void Form1_Load(object sender, EventArgs e)
{

}
```

Invocare il metodo:

```
EventName(SenderObject, EventArguments);
```

Dichiarazione del gestore eventi anonimo

Dichiarazione di evento:

```
public event EventHandler<EventArgsType> EventName;
```

Dichiarazione del gestore eventi che utilizza l' operatore lambda => e si iscrive all'evento:

```
EventName += (obj, eventArgs) => { /* Handler logic */ };
```

Dichiarazione del gestore eventi che utilizza la sintassi del metodo anonimo delegato :

```
EventName += delegate(object obj, EventArgsType eventArgs) { /* Handler Logic */ };
```

Dichiarazione e sottoscrizione di un gestore di eventi che non utilizza il parametro dell'evento e

pertanto può utilizzare la sintassi precedente senza dover specificare i parametri:

```
EventName += delegate { /* Handler Logic */ }
```

Invocazione dell'evento:

```
EventName?.Invoke(SenderObject, EventArguments);
```

Dichiarazione di eventi non standard

Gli eventi possono essere di qualsiasi tipo di delegato, non solo `EventHandler` e `EventHandler<T>` . Per esempio:

```
//Declaring an event
public event Action<Param1Type, Param2Type, ...> EventName;
```

Questo è usato in modo simile agli eventi `EventHandler` standard:

```
//Adding a named event handler
public void HandlerName(Param1Type parameter1, Param2Type parameter2, ...) {
    /* Handler logic */
}
EventName += HandlerName;

//Adding an anonymous event handler
EventName += (parameter1, parameter2, ...) => { /* Handler Logic */ };

//Invoking the event
EventName(parameter1, parameter2, ...);
```

È possibile dichiarare più eventi dello stesso tipo in una singola istruzione, in modo simile ai campi e alle variabili locali (sebbene ciò possa spesso essere una cattiva idea):

```
public event EventHandler Event1, Event2, Event3;
```

Questo dichiara tre eventi separati (`Event1` , `Event2` ed `Event3`) tutti di tipo `EventHandler` .
Nota: sebbene alcuni compilatori possano accettare questa sintassi sia nelle interfacce che nelle classi, la specifica C # (v5.0 §13.2.3) fornisce la grammatica per le interfacce che non lo consentono, quindi l'utilizzo di questo nelle interfacce potrebbe non essere affidabile con diversi compilatori.

Creazione di eventi personalizzati contenenti dati aggiuntivi

Gli eventi personalizzati di solito richiedono argomenti di eventi personalizzati contenenti informazioni sull'evento. Ad esempio `MouseEventArgs` utilizzato dagli eventi del mouse come `MouseDown` o `MouseUp` , contiene informazioni sulla `Location` o sui `Buttons` utilizzati per generare l'evento.

Quando si creano nuovi eventi, per creare un evento personalizzato arg:

- Creare una classe derivante da `EventArgs` e definire le proprietà per i dati necessari.
- Come convenzione, il nome della classe dovrebbe terminare con `EventArgs` .

Esempio

Nell'esempio seguente, creiamo un evento `PriceChangingEventArgs` per la proprietà `Price` di una classe. La classe di dati evento contiene un `CurrentPrice` e `NewPrice` . L'evento aumenta quando si assegna un nuovo valore alla proprietà `Price` e si fa sapere al consumatore che il valore sta cambiando e consente loro di conoscere il prezzo corrente e il nuovo prezzo:

PriceChangingEventArgs

```
public class PriceChangingEventArgs : EventArgs
{
    public PriceChangingEventArgs(int currentPrice, int newPrice)
    {
        this.CurrentPrice = currentPrice;
        this.NewPrice = newPrice;
    }

    public int CurrentPrice { get; private set; }
    public int NewPrice { get; private set; }
}
```

Prodotto

```
public class Product
{
    public event EventHandler<PriceChangingEventArgs> PriceChanging;

    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(price, value);
            OnPriceChanging(e);
            price = value;
        }
    }

    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PriceChanging;
        if (handler != null)
            handler(this, e);
    }
}
```

È possibile migliorare l'esempio consentendo al consumatore di modificare il nuovo valore e quindi il valore verrà utilizzato per la proprietà. Per fare ciò è sufficiente applicare questi cambiamenti nelle classi.

Modifica la definizione di `NewPrice` per essere impostabile:

```
public int NewPrice { get; set; }
```

Modifica la definizione di `Price` per utilizzare `e.NewPrice` come valore della proprietà, dopo aver chiamato `OnPriceChanging` :

```
int price;
public int Price
{
    get { return price; }
    set
    {
        var e = new PriceChangingEventArgs(price, value);
        OnPriceChanging(e);
        price = e.NewPrice;
    }
}
```

Creare un evento cancellabile

Un evento cancellabile può essere generato da una classe quando sta per eseguire un'azione che può essere annullata, come ad esempio l'evento `FormClosing` di un `Form` .

Per creare un evento del genere:

- Creare un nuovo argomento evento derivato da `CancelEventArgs` e aggiungere proprietà aggiuntive per i dati degli eventi.
- Crea un evento utilizzando `EventHandler<T>` e utilizza la nuova classe di evento event cancel che hai creato.

Esempio

Nell'esempio seguente, creiamo un evento `PriceChangingEventArgs` per la proprietà `Price` di una classe. La classe di dati dell'evento contiene un `Value` che consente al consumatore di conoscere il nuovo. L'evento aumenta quando si assegna un nuovo valore alla proprietà `Price` e si consente al consumatore di sapere che il valore sta cambiando e di consentire loro di annullare l'evento. Se il consumatore annulla l'evento, verrà utilizzato il valore precedente per `Price` :

PriceChangingEventArgs

```
public class PriceChangingEventArgs : CancelEventArgs
{
    int value;
    public int Value
    {
        get { return value; }
    }
    public PriceChangingEventArgs(int value)
    {
        this.value = value;
    }
```

```
}
```

Prodotto

```
public class Product
{
    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(value);
            OnPriceChanging(e);
            if (!e.Cancel)
                price = value;
        }
    }

    public event EventHandler<PriceChangingEventArgs> PropertyChanging;
    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PropertyChanging;
        if (handler != null)
            PropertyChanging(this, e);
    }
}
```

Proprietà dell'evento

Se una classe aumenta di molto il numero di eventi, il costo di archiviazione di un campo per delegato potrebbe non essere accettabile. .NET Framework fornisce le proprietà degli eventi per questi casi. In questo modo è possibile utilizzare un'altra struttura dati come `EventHandlerList` per memorizzare i delegati dell'evento:

```
public class SampleClass
{
    // Define the delegate collection.
    protected EventHandlerList eventDelegates = new EventHandlerList();

    // Define a unique key for each event.
    static readonly object someEventKey = new object();

    // Define the SomeEvent event property.
    public event EventHandler SomeEvent
    {
        add
        {
            // Add the input delegate to the collection.
            eventDelegates.AddHandler(someEventKey, value);
        }
        remove
        {
            // Remove the input delegate from the collection.
            eventDelegates.RemoveHandler(someEventKey, value);
        }
    }
```

```
    // Raise the event with the delegate specified by someEventKey
    protected void OnSomeEvent(EventArgs e)
    {
        var handler = (EventHandler)eventDelegates[someEventKey];
        if (handler != null)
            handler(this, e);
    }
}
```

Questo approccio è ampiamente utilizzato in framework GUI come WinForms in cui i controlli possono avere dozzine e persino centinaia di eventi.

Notare che `EventHandlerList` non è thread-safe, quindi se si prevede che la classe venga utilizzata da più thread, sarà necessario aggiungere istruzioni di blocco o altri meccanismi di sincronizzazione (o utilizzare una memoria che fornisce sicurezza di thread).

Leggi eventi online: https://riptutorial.com/it/csharp/topic/64/eventi

Capitolo 51: File e streaming I / O

introduzione

Gestisce i file.

Sintassi

- `new System.IO.StreamWriter(string path)`
- `new System.IO.StreamWriter(string path, bool append)`
- `System.IO.StreamWriter.WriteLine(string text)`
- `System.IO.StreamWriter.WriteAsync(string text)`
- `System.IO.Stream.Close()`
- `System.IO.File.ReadAllText(string path)`
- `System.IO.File.ReadAllLines(string path)`
- `System.IO.File.ReadLines(string path)`
- `System.IO.File.WriteAllText(string path, string text)`
- `System.IO.File.WriteAllLines(string path, IEnumerable<string> contents)`
- `System.IO.File.Copy(string source, string dest)`
- `System.IO.File.Create(string path)`
- `System.IO.File.Delete(string path)`
- `System.IO.File.Move(string source, string dest)`
- `System.IO.Directory.GetFiles(string path)`

Parametri

Parametro	Dettagli
sentiero	La posizione del file.
aggiungere	Se il file esiste, true aggiungerà i dati alla fine del file (append), false sovrascriverà il file.
testo	Testo da scrivere o archiviare.
contenuto	Una raccolta di stringhe da scrivere.
fonte	La posizione del file che si desidera utilizzare.
dest	La posizione in cui si desidera un file.

Osservazioni

- Assicurati sempre di chiudere gli oggetti `Stream` . Questo può essere fatto con un blocco `using` come mostrato sopra o chiamando manualmente `myStream.Close()` .
- Assicurarsi che l'utente corrente disponga delle autorizzazioni necessarie sul percorso in cui

si sta tentando di creare il file.
- Le stringhe di Verbatim devono essere utilizzate quando si dichiara una stringa di percorso che include barre retroverse, ad esempio: `@"C:\MyFolder\MyFile.txt"`

Examples

Lettura da un file usando la classe System.IO.File

È possibile utilizzare la funzione **System.IO.File.ReadAllText** per leggere l'intero contenuto di un file in una stringa.

```
string text = System.IO.File.ReadAllText(@"C:\MyFolder\MyTextFile.txt");
```

Puoi anche leggere un file come una matrice di linee usando la funzione **System.IO.File.ReadAllLines** :

```
string[] lines = System.IO.File.ReadAllLines(@"C:\MyFolder\MyTextFile.txt");
```

Scrittura di righe su un file utilizzando la classe System.IO.StreamWriter

La classe **System.IO.StreamWriter** :

> Implementa un TextWriter per scrivere caratteri su uno stream in una particolare codifica.

Utilizzando il metodo `WriteLine` , è possibile scrivere il contenuto riga per riga in un file.

Si noti l'uso della parola chiave `using` che si assicura che l'oggetto StreamWriter sia eliminato non appena esce dall'ambito e quindi il file viene chiuso.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt"))
{
    foreach (string line in lines)
    {
        sw.WriteLine(line);
    }
}
```

Si noti che StreamWriter può ricevere un secondo parametro `bool` nel suo costruttore, consentendo di `Append` a un file invece di sovrascrivere il file:

```
bool appendExistingFile = true;
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt",
appendExistingFile ))
{
    sw.WriteLine("This line will be appended to the existing file");
}
```

Scrivere su un file usando la classe System.IO.File

È possibile utilizzare la funzione **System.IO.File.WriteAllText** per scrivere una stringa in un file.

```
string text = "String that will be stored in the file";
System.IO.File.WriteAllText(@"C:\MyFolder\OutputFile.txt", text);
```

È anche possibile utilizzare la funzione **System.IO.File.WriteAllLines** che riceve un oggetto `IEnumerable<String>` come secondo parametro (anziché una singola stringa nell'esempio precedente). Ciò ti consente di scrivere contenuti da una serie di linee.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
System.IO.File.WriteAllLines(@"C:\MyFolder\OutputFile.txt", lines);
```

Lettura lenta di un file riga per riga tramite un oggetto IEnumerable

Quando si lavora con file di grandi dimensioni, è possibile utilizzare il metodo `System.IO.File.ReadLines` per leggere tutte le righe da un file in un oggetto `IEnumerable<string>` . Questo è simile a `System.IO.File.ReadAllLines` , tranne per il fatto che non carica l'intero file in memoria in una volta, rendendolo più efficiente quando si lavora con file di grandi dimensioni.

```
IEnumerable<string> AllLines = File.ReadLines("file_name.txt", Encoding.Default);
```

Il secondo parametro di File.ReadLines è facoltativo. È possibile utilizzarlo quando è necessario specificare la codifica.

È importante notare che chiamare `ToArray` , `ToList` o un'altra funzione simile costringerà tutte le linee a essere caricate contemporaneamente, il che significa che il vantaggio dell'utilizzo di `ReadLines` è annullato. È meglio enumerare su `IEnumerable` utilizzando un ciclo `foreach` o LINQ se si utilizza questo metodo.

Crea file

Classe statica di file

Usando il metodo `Create` della classe statica `File` possiamo creare file. Il metodo crea il file nel percorso specificato, allo stesso tempo apre il file e ci dà il `FileStream` del file. Assicurati di chiudere il file dopo aver finito con esso.

EX1:

```
var fileStream1 = File.Create("samplePath");
/// you can write to the fileStream1
fileStream1.Close();
```

EX2:

```
using(var fileStream1 = File.Create("samplePath"))
```

```
{
    /// you can write to the fileStream1
}
```

EX3:

```
File.Create("samplePath").Close();
```

Classe FileStream

Ci sono molti sovraccarichi di questo costruttore di classi che è in realtà ben documentato qui . Di seguito l'esempio è per quello che copre le funzionalità più utilizzate di questa classe.

```
var fileStream2 = new FileStream("samplePath", FileMode.OpenOrCreate, FileAccess.ReadWrite,
FileShare.None);
```

È possibile controllare l'enumerazione per FileMode , FileAccess e FileShare da tali collegamenti. Ciò che in pratica significa sono i seguenti:

FileMode: risposte "Se il file deve essere creato? Aperto? Creare se non esiste, quindi aprire?" un po 'domande.

FileAccess: Answers "Devo essere in grado di leggere il file, scrivere sul file o entrambi?" un po 'domande.

FileShare: Answers "Gli altri utenti possono leggere, scrivere, ecc. Sul file mentre lo sto usando simultaneamente?" un po 'domande.

Copia il file

Classe statica di file

`File` classe statica dei `File` può essere facilmente utilizzata per questo scopo.

```
File.Copy(@"sourcePath\abc.txt", @"destinationPath\abc.txt");
File.Copy(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Osservazione: con questo metodo, il file viene copiato, il che significa che verrà letto dall'origine e quindi scritto nel percorso di destinazione. Questo è un processo che consuma risorse, richiede tempo relativamente alla dimensione del file e può bloccare il programma se non si utilizzano i thread.

Sposta il file

Classe statica di file

La classe statica del file può essere facilmente utilizzata per questo scopo.

```
File.Move(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Nota 1: modifica solo l'indice del file (se il file viene spostato nello stesso volume). Questa operazione non richiede tempo relativamente alla dimensione del file.

Remark2: impossibile eseguire l'override di un file esistente sul percorso di destinazione.

Cancella il file

```
string path = @"c:\path\to\file.txt";
File.Delete(path);
```

Mentre `Delete` non genera un'eccezione se il file non esiste, genererà un'eccezione, ad esempio se il percorso specificato non è valido o se il chiamante non dispone delle autorizzazioni richieste. Dovresti sempre inserire le chiamate su `Delete` nel blocco try-catch e gestire tutte le eccezioni previste. In caso di condizioni di gara possibili, avvolgere la logica all'interno dell'istruzione di blocco .

File e directory

Ottieni tutti i file nella directory

```
var FileSearchRes = Directory.GetFiles(@Path, "*.*", SearchOption.AllDirectories);
```

Restituisce un array di `FileInfo` , che rappresenta tutti i file nella directory specificata.

Ottieni file con estensione specifica

```
var FileSearchRes = Directory.GetFiles(@Path, "*.pdf", SearchOption.AllDirectories);
```

Restituisce un array di `FileInfo` , che rappresenta tutti i file nella directory specificata con l'estensione specificata.

Async scrive il testo in un file usando StreamWriter

```
// filename is a string with the full path
// true is to append
using (System.IO.StreamWriter file = new System.IO.StreamWriter(filename, true))
{
   // Can write either a string or char array
   await file.WriteAsync(text);
}
```

Leggi File e streaming I / O online: https://riptutorial.com/it/csharp/topic/4266/file-e-streaming-i---o

Capitolo 52: FileSystemWatcher

Sintassi

- public FileSystemWatcher ()
- public FileSystemWatcher (percorso stringa)
- public FileSystemWatcher (percorso stringa, filtro stringa)

Parametri

sentiero	filtro
La directory da monitorare, nella notazione standard o Universal Naming Convention (UNC).	Il tipo di file da guardare. Ad esempio, "*.txt" controlla le modifiche a tutti i file di testo.

Examples

FileWatcher di base

L'esempio seguente crea un `FileSystemWatcher` per guardare la directory specificata in fase di esecuzione. Il componente è impostato per controllare le modifiche in **LastWrite** e **LastAccess** time, la creazione, la cancellazione o la ridenominazione dei file di testo nella directory. Se un file viene modificato, creato o eliminato, il percorso del file viene stampato sulla console. Quando un file viene rinominato, i percorsi vecchi e nuovi vengono stampati sulla console.

Utilizzare gli spazi dei nomi System.Diagnostics e System.IO per questo esempio.

```
FileSystemWatcher watcher;

private void watch()
{
  // Create a new FileSystemWatcher and set its properties.
  watcher = new FileSystemWatcher();
  watcher.Path = path;

 /* Watch for changes in LastAccess and LastWrite times, and
       the renaming of files or directories. */
  watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
                       | NotifyFilters.FileName | NotifyFilters.DirectoryName;

  // Only watch text files.
  watcher.Filter = "*.txt*";

  // Add event handler.
  watcher.Changed += new FileSystemEventHandler(OnChanged);
  // Begin watching.
  watcher.EnableRaisingEvents = true;
```

```
}

// Define the event handler.
private void OnChanged(object source, FileSystemEventArgs e)
{
  //Copies file to another directory or another action.
  Console.WriteLine("File: " +  e.FullPath + " " + e.ChangeType);
}
```

IsFileReady

Un errore comune che molte persone iniziano con FileSystemWatcher non sta prendendo in considerazione che l'evento FileWatcher viene generato non appena viene creato il file. Tuttavia, potrebbe essere necessario del tempo per il completamento del file.

Esempio :

Ad esempio, prendi una dimensione del file di 1 GB. Il file apr ask creato da un altro programma (Explorer.exe lo copia da qualche parte) ma ci vorranno alcuni minuti per terminare quel processo. L'evento aumenta il tempo di creazione e devi aspettare che il file sia pronto per essere copiato.

Questo è un metodo per verificare se il file è pronto.

```
 public static bool IsFileReady(String sFilename)
{
    // If the file can be opened for exclusive access it means that the file
    // is no longer locked by another process.
    try
    {
        using (FileStream inputStream = File.Open(sFilename, FileMode.Open, FileAccess.Read,
FileShare.None))
        {
            if (inputStream.Length > 0)
            {
                return true;
            }
            else
            {
                return false;
            }

        }
    }
    catch (Exception)
    {
        return false;
    }
}
```

Leggi FileSystemWatcher online: https://riptutorial.com/it/csharp/topic/5061/filesystemwatcher

Capitolo 53: Filtri di azione

Examples

Filtri di azione personalizzati

Scriviamo filtri di azione personalizzati per vari motivi. Potremmo avere un filtro azioni personalizzato per la registrazione o per salvare i dati nel database prima di qualsiasi esecuzione di azioni. Potremmo anche averne uno per prelevare i dati dal database e impostarlo come valori globali dell'applicazione.

Per creare un filtro azioni personalizzato, è necessario eseguire le seguenti attività:

1. Crea una classe
2. Eredita dalla classe ActionFilterAttribute

Sostituire almeno uno dei seguenti metodi:

OnActionExecuting : questo metodo viene chiamato prima che venga eseguita un'azione del controllore.

OnActionExecuted : questo metodo viene chiamato dopo l'esecuzione di un'azione del controller.

OnResultExecuting : questo metodo viene chiamato prima dell'esecuzione di un risultato dell'azione del controller.

OnResultExecuted : questo metodo viene chiamato dopo l'esecuzione di un risultato dell'azione del controller.

Il filtro può essere creato come mostrato nell'elenco seguente:

```
using System;

using System.Diagnostics;

using System.Web.Mvc;



namespace WebApplication1
{

    public class MyFirstCustomFilter : ActionFilterAttribute
    {
        public override void OnResultExecuting(ResultExecutingContext filterContext)
        {
            //You may fetch data from database here
            filterContext.Controller.ViewBag.GreetMesssage = "Hello Foo";
            base.OnResultExecuting(filterContext);
        }
```

```
            public override void OnActionExecuting(ActionExecutingContext filterContext)
            {
                var controllerName = filterContext.RouteData.Values["controller"];
                var actionName = filterContext.RouteData.Values["action"];
                var message = String.Format("{0} controller:{1} action:{2}",
"onactionexecuting", controllerName, actionName);
                Debug.WriteLine(message, "Action Filter Log");
                base.OnActionExecuting(filterContext);
            }
        }
    }
```

Leggi Filtri di azione online: https://riptutorial.com/it/csharp/topic/1505/filtri-di-azione

Capitolo 54: Func delegati

Sintassi

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

Parametri

Parametro	Dettagli
`arg` o `arg1`	il (primo) parametro del metodo
`arg2`	il secondo parametro del metodo
`arg3`	il terzo parametro del metodo
`arg4`	il quarto parametro del metodo
`T` o `T1`	il tipo del (primo) parametro del metodo
`T2`	il tipo del secondo parametro del metodo
`T3`	il tipo del terzo parametro del metodo
`T4`	il tipo del quarto parametro del metodo
`TResult`	il tipo di ritorno del metodo

Examples

Senza parametri

Questo esempio mostra come creare un delegato che incapsula il metodo che restituisce l'ora corrente

```
static DateTime UTCNow()
{
    return DateTime.UtcNow;
}

static DateTime LocalNow()
{
    return DateTime.Now;
}
```

```
static void Main(string[] args)
{
    Func<DateTime> method = UTCNow;
    // method points to the UTCNow method
    // that retuns current UTC time
    DateTime utcNow = method();

    method = LocalNow;
    // now method points to the LocalNow method
    // that returns local time

    DateTime localNow = method();
}
```

Con più variabili

```
static int Sum(int a, int b)
{
    return a + b;
}

static int Multiplication(int a, int b)
{
    return a * b;
}

static void Main(string[] args)
{
    Func<int, int, int> method = Sum;
    // method points to the Sum method
    // that retuns 1 int variable and takes 2 int variables
    int sum = method(1, 1);

    method = Multiplication;
    // now method points to the Multiplication method

    int multiplication = method(1, 1);
}
```

Lambda e metodi anonimi

Un metodo anonimo può essere assegnato ovunque sia previsto un delegato:

```
Func<int, int> square = delegate (int x) { return x * x; }
```

Le espressioni Lambda possono essere utilizzate per esprimere la stessa cosa:

```
Func<int, int> square = x => x * x;
```

In entrambi i casi, ora possiamo invocare il metodo memorizzato all'interno di un `square` come questo:

```
var sq = square.Invoke(2);
```

O come una stenografia:

```
var sq = square(2);
```

Si noti che per l'assegnazione del tipo sicuro, i tipi di parametri e il tipo restituito del metodo anonimo devono corrispondere a quelli del tipo delegato:

```
Func<int, int> sum = delegate (int x, int y) { return x + y; } // error
Func<int, int> sum = (x, y) => x + y; // error
```

Parametri di tipo covariant e controvariante

`Func` supporta anche Covariant & Contravariant

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }

class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Leggi Func delegati online: https://riptutorial.com/it/csharp/topic/2769/func-delegati

Capitolo 55: Funzione con più valori di ritorno

Osservazioni

Non c'è una risposta inerente in C # a questo - così chiamato - bisogno. Tuttavia ci sono soluzioni alternative per soddisfare questa esigenza.

Il motivo per cui qualificano il bisogno come "così chiamato" è che abbiamo bisogno solo di metodi con 2 o più di 2 valori da restituire quando violiamo i principi di programmazione. Soprattutto il principio della singola responsabilità .

Quindi, sarebbe meglio essere avvisati quando abbiamo bisogno di funzioni che restituiscono 2 o più valori e migliorare il nostro design.

Examples

soluzione "oggetto anonimo" + "parola chiave dinamica"

Puoi restituire un oggetto anonimo dalla tua funzione

```
public static object FunctionWithUnknowReturnValues ()
{
    /// anonymous object
    return new { a = 1, b = 2 };
}
```

E assegna il risultato a un oggetto dinamico e leggi i valori in esso contenuti.

```
/// dynamic object
dynamic x = FunctionWithUnknowReturnValues();

Console.WriteLine(x.a);
Console.WriteLine(x.b);
```

Soluzione tuple

Puoi restituire un'istanza della classe `Tuple` dalla tua funzione con due parametri del modello come `Tuple<string, MyClass>` :

```
public Tuple<string, MyClass> FunctionWith2ReturnValues ()
{
    return Tuple.Create("abc", new MyClass());
}
```

E leggi i valori come di seguito:

```
Console.WriteLine(x.Item1);
Console.WriteLine(x.Item2);
```

Parametri Ref e Out

La parola chiave `ref` viene utilizzata per passare un argomento come riferimento . `out` farà lo stesso di `ref` ma non richiede un valore assegnato dal chiamante prima di chiamare la funzione.

Parametro Ref : -Se si desidera passare una variabile come parametro ref, è necessario inizializzarla prima di passarla come parametro ref al metodo.

Out Parameter: - Se si desidera passare una variabile come parametro out, non è necessario inizializzarla prima di passarla come parametro out al metodo.

```
static void Main(string[] args)
{
    int a = 2;
    int b = 3;
    int add = 0;
    int mult= 0;
    AddOrMult(a, b, ref add, ref mult); //AddOrMult(a, b, out add, out mult);
    Console.WriteLine(add); //5
    Console.WriteLine(mult); //6
}

private static void AddOrMult(int a, int b, ref int add, ref int mult) //AddOrMult(int a, int
b, out int add, out int mult)
{
    add = a + b;
    mult = a * b;
}
```

Leggi Funzione con più valori di ritorno online: https://riptutorial.com/it/csharp/topic/3908/funzione-con-piu-valori-di-ritorno

Capitolo 56: Funzioni hash

Osservazioni

MD5 e SHA1 non sono sicuri e dovrebbero essere evitati. Gli esempi esistono per scopi didattici e perché il software legacy può ancora utilizzare questi algoritmi.

Examples

MD5

Le funzioni hash associano le stringhe binarie di una lunghezza arbitraria a stringhe binarie di una lunghezza fissa.

L'algoritmo `MD5` è una funzione hash ampiamente utilizzata che produce un valore hash a 128 bit (16 byte, 32 caratteri esadecimali).

Il metodo `ComputeHash` della classe `System.Security.Cryptography.MD5` restituisce l'hash come una matrice di 16 byte.

Esempio:

```
using System;
using System.Security.Cryptography;
using System.Text;

internal class Program
{
    private static void Main()
    {
        var source = "Hello World!";

        // Creates an instance of the default implementation of the MD5 hash algorithm.
        using (var md5Hash = MD5.Create())
        {
            // Byte array representation of source string
            var sourceBytes = Encoding.UTF8.GetBytes(source);

            // Generate hash value(Byte Array) for input data
            var hashBytes = md5Hash.ComputeHash(sourceBytes);

            // Convert hash byte array to string
            var hash = BitConverter.ToString(hashBytes).Replace("-", string.Empty);

            // Output the MD5 hash
            Console.WriteLine("The MD5 hash of " + source + " is: " + hash);
        }
    }
}
```

Output: l'hash MD5 di Hello World! è: ED076287532E86365E841E92BFC50D8C

Problemi di sicurezza:

Come la maggior parte delle funzioni di hash, MD5 non è né crittografia né codifica. Può essere invertito con un attacco a forza bruta e soffre di ampie vulnerabilità contro gli attacchi di collisione e di preimage.

SHA1

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA1 sha1Hash = SHA1.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha1Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-",String.Empty);

                Console.WriteLine("The SHA1 hash of " + source + " is: " + hash);
            }
        }
    }
 }
```

Produzione:

L'hash SHA1 di Hello Word! è: 2EF7BDE608CE5404E97D5F042F95F89F1C232871

SHA256

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA256 sha256Hash = SHA256.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
```

```
                byte[] hashBytes = sha256Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA256 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Produzione:

L'hash SHA256 di Hello World! è:
7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069

SHA384

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA384 sha384Hash = SHA384.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha384Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA384 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Produzione:

L'hash SHA384 di Hello World! is:
BFD76C0EBBD006FEE583410547C1887B0292BE76D582D96C242D2A792723E3FD6FD061F9D5CFD

SHA512

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA512 sha512Hash = SHA512.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha512Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA512 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Uscita: l'hash SHA512 di Hello World! è:
861844D6704E8573FEC34D967E20BCFEF3D424CF48BE04E6DC08F2BD58C729743371015EAD891C

PBKDF2 per l'hashing delle password

PBKDF2 ("Funzione di derivazione chiave basata su password 2") è una delle funzioni hash consigliate per l'hashing delle password. Fa parte di rfc-2898 .

.NET's `Rfc2898DeriveBytes` -Class è basato su HMACSHA1.

```
    using System.Security.Cryptography;

    ...

    public const int SALT_SIZE = 24; // size in bytes
    public const int HASH_SIZE = 24; // size in bytes
    public const int ITERATIONS = 100000; // number of pbkdf2 iterations

    public static byte[] CreateHash(string input)
    {
        // Generate a salt
        RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
        byte[] salt = new byte[SALT_SIZE];
        provider.GetBytes(salt);

        // Generate the hash
        Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(input, salt, ITERATIONS);
        return pbkdf2.GetBytes(HASH_SIZE);
    }
```

PBKDF2 richiede un sale e il numero di iterazioni.

iterazioni:

Un numero elevato di iterazioni rallenterà l'algoritmo, il che rende molto più difficile il crack delle password. Si raccomanda quindi un numero elevato di iterazioni. Ad esempio, PBKDF2 è un ordine di grandezza più lento di MD5.

Sale:

Un salt impedirà la ricerca di valori hash nelle tabelle arcobaleno. Deve essere memorizzato insieme all'hash della password. Si consiglia un sale per password (non un sale globale).

Completa password Hashing Solution utilizzando Pbkdf2

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace YourCryptoNamespace
{
    /// <summary>
    /// Salted password hashing with PBKDF2-SHA1.
  /// Compatibility: .NET 3.0 and later.
  /// </summary>
  /// <remarks>See http://crackstation.net/hashing-security.htm for much more on password
hashing.</remarks>
  public static class PasswordHashProvider
  {
    /// <summary>
    /// The salt byte size, 64 length ensures safety but could be increased / decreased
    /// </summary>
    private const int SaltByteSize = 64;
    /// <summary>
    /// The hash byte size,
    /// </summary>
    private const int HashByteSize = 64;
    /// <summary>
    ///  High iteration count is less likely to be cracked
    /// </summary>
    private const int Pbkdf2Iterations = 10000;

    /// <summary>
    /// Creates a salted PBKDF2 hash of the password.
    /// </summary>
    /// <remarks>
    /// The salt and the hash have to be persisted side by side for the password. They could
be persisted as bytes or as a string using the convenience methods in the next class to
convert from byte[] to string and later back again when executing password validation.
    /// </remarks>
    /// <param name="password">The password to hash.</param>
    /// <returns>The hash of the password.</returns>
    public static PasswordHashContainer CreateHash(string password)
    {
      // Generate a random salt
      using (var csprng = new RNGCryptoServiceProvider())
      {
        // create a unique salt for every password hash to prevent rainbow and dictionary
based attacks
        var salt = new byte[SaltByteSize];
        csprng.GetBytes(salt);

        // Hash the password and encode the parameters
        var hash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);

        return new PasswordHashContainer(hash, salt);
      }
    }
    /// <summary>
```

```
    /// Recreates a password hash based on the incoming password string and the stored salt
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The salt existing.</param>
    /// <returns>the generated hash based on the password and salt</returns>
    public static byte[] CreateHash(string password, byte[] salt)
    {
      // Extract the parameters from the hash
      return Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
    }

    /// <summary>
    /// Validates a password given a hash of the correct one.
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The existing stored salt.</param>
    /// <param name="correctHash">The hash of the existing password.</param>
    /// <returns><c>true</c> if the password is correct. <c>false</c> otherwise. </returns>
    public static bool ValidatePassword(string password, byte[] salt, byte[] correctHash)
    {
      // Extract the parameters from the hash
      byte[] testHash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
      return CompareHashes(correctHash, testHash);
    }
    /// <summary>
    /// Compares two byte arrays (hashes)
    /// </summary>
    /// <param name="array1">The array1.</param>
    /// <param name="array2">The array2.</param>
    /// <returns><c>true</c> if they are the same, otherwise <c>false</c></returns>
    public static bool CompareHashes(byte[] array1, byte[] array2)
    {
      if (array1.Length != array2.Length) return false;
      return !array1.Where((t, i) => t != array2[i]).Any();
    }

    /// <summary>
    /// Computes the PBKDF2-SHA1 hash of a password.
    /// </summary>
    /// <param name="password">The password to hash.</param>
    /// <param name="salt">The salt.</param>
    /// <param name="iterations">The PBKDF2 iteration count.</param>
    /// <param name="outputBytes">The length of the hash to generate, in bytes.</param>
    /// <returns>A hash of the password.</returns>
    private static byte[] Pbkdf2(string password, byte[] salt, int iterations, int
outputBytes)
    {
      using (var pbkdf2 = new Rfc2898DeriveBytes(password, salt))
      {
        pbkdf2.IterationCount = iterations;
        return pbkdf2.GetBytes(outputBytes);
      }
    }
  }

  /// <summary>
  /// Container for password hash and salt and iterations.
  /// </summary>
  public sealed class PasswordHashContainer
  {
    /// <summary>
```

```
    /// Gets the hashed password.
    /// </summary>
    public byte[] HashedPassword { get; private set; }
    /// <summary>
    /// Gets the salt.
    /// </summary>
    public byte[] Salt { get; private set; }

    /// <summary>
    /// Initializes a new instance of the <see cref="PasswordHashContainer" /> class.
    /// </summary>
    /// <param name="hashedPassword">The hashed password.</param>
    /// <param name="salt">The salt.</param>
    public PasswordHashContainer(byte[] hashedPassword, byte[] salt)
    {
      this.HashedPassword = hashedPassword;
      this.Salt = salt;
    }
  }

  /// <summary>
  /// Convenience methods for converting between hex strings and byte array.
  /// </summary>
  public static class ByteConverter
  {
    /// <summary>
    /// Converts the hex representation string to an array of bytes
    /// </summary>
    /// <param name="hexedString">The hexed string.</param>
    /// <returns></returns>
    public static byte[] GetHexBytes(string hexedString)
    {
      var bytes = new byte[hexedString.Length / 2];
      for (var i = 0; i < bytes.Length; i++)
      {
        var strPos = i * 2;
        var chars = hexedString.Substring(strPos, 2);
        bytes[i] = Convert.ToByte(chars, 16);
      }
      return bytes;
    }
    /// <summary>
    /// Gets a hex string representation of the byte array passed in.
    /// </summary>
    /// <param name="bytes">The bytes.</param>
    public static string GetHexString(byte[] bytes)
    {
      return BitConverter.ToString(bytes).Replace("-", "").ToUpper();
    }
  }
}

/*
 * Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).
 * Copyright (c) 2013, Taylor Hornby
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
```

```
* this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright notice,
* this list of conditions and the following disclaimer in the documentation
* and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*/
```

Si prega di consultare questa eccellente risorsa Crackstation - Salted Password Hashing - Farlo bene per ulteriori informazioni. Parte di questa soluzione (la funzione di hashing) era basata sul codice di quel sito.

Leggi Funzioni hash online: https://riptutorial.com/it/csharp/topic/2774/funzioni-hash

Capitolo 57: Garbage Collector in .Net

Examples

Compattazione del mucchio di oggetti di grandi dimensioni

Di default il Large Object Heap non è compattato a differenza del classico Object Heap che può portare alla frammentazione della memoria e, inoltre, può portare a `OutOfMemoryException` s

A partire da .NET 4.5.1 esiste un'opzione per comprimere esplicitamente l'heap di oggetti di grandi dimensioni (insieme a una garbage collection):

```
GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

Proprio come qualsiasi richiesta di garbage collection esplicita (si chiama richiesta perché CLR non è obbligato a condurla) usa con cautela e, per impostazione predefinita, evitala se puoi perché può `GC` statistiche di `GC` , diminuendo le sue prestazioni.

Riferimenti deboli

In .NET, il GC alloca gli oggetti quando non ci sono riferimenti a loro lasciati. Pertanto, mentre un oggetto può ancora essere raggiunto dal codice (c'è un forte riferimento ad esso), il GC non assegnerà questo oggetto. Questo può diventare un problema se ci sono molti oggetti di grandi dimensioni.

Un riferimento debole è un riferimento che consente al GC di raccogliere l'oggetto pur consentendo l'accesso all'oggetto. Un riferimento debole è valido solo durante l'intervallo di tempo indeterminato finché l'oggetto non viene raccolto quando non esistono riferimenti forti. Quando si utilizza un riferimento debole, l'applicazione può ancora ottenere un riferimento forte all'oggetto, che ne impedisce la raccolta. Quindi i riferimenti deboli possono essere utili per conservare oggetti di grandi dimensioni che sono costosi da inizializzare, ma dovrebbero essere disponibili per la raccolta dei dati inutili se non sono attivamente utilizzati.

Uso semplice:

```
WeakReference reference = new WeakReference(new object(), false);

GC.Collect();

object target = reference.Target;
if (target != null)
  DoSomething(target);
```

Quindi riferimenti deboli potrebbero essere usati per mantenere, per esempio, una cache di oggetti. Tuttavia, è importante ricordare che c'è sempre il rischio che il garbage collector raggiunga l'oggetto prima che venga ristabilito un riferimento forte.

I riferimenti deboli sono anche utili per evitare perdite di memoria. Un tipico caso d'uso è con gli eventi.

Supponiamo di avere qualche gestore di un evento su una fonte:

```
Source.Event += new EventHandler(Handler)
```

Questo codice registra un gestore di eventi e crea un riferimento forte dall'origine evento all'oggetto in ascolto. Se l'oggetto di origine ha una durata maggiore rispetto al listener e il listener non ha più bisogno dell'evento quando non ci sono altri riferimenti ad esso, l'uso di eventi .NET normali causa una perdita di memoria: l'oggetto di origine contiene oggetti listener in memoria che dovrebbe essere raccolta dei rifiuti

In questo caso, potrebbe essere una buona idea usare il modello di evento debole .

Qualcosa di simile a:

```
public static class WeakEventManager
    {
    public static void SetHandler<S, TArgs>(
    Action<EventHandler<TArgs>> add,
    Action<EventHandler<TArgs>> remove,
    S subscriber,
    Action<S, TArgs> action)
    where TArgs : EventArgs
    where S : class
        {
            var subscrWeakRef = new WeakReference(subscriber);
            EventHandler<TArgs> handler = null;

            handler = (s, e) =>
            {
                var subscrStrongRef = subscrWeakRef.Target as S;
                if (subscrStrongRef != null)
                {
                    action(subscrStrongRef, e);
                }
                else
                {
                    remove(handler);
                    handler = null;
                }
            };

            add(handler);
        }
    }
```

e usato in questo modo:

```
 EventSource s = new EventSource();
 Subscriber subscriber = new Subscriber();
 WeakEventManager.SetHandler<Subscriber, SomeEventArgs>(a => s.Event += a, r => s.Event -= r,
subscriber, (s,e) => { s.HandleEvent(e); });
```

In questo caso ovviamente abbiamo alcune restrizioni: l'evento deve essere a

```
public event EventHandler<SomeEventArgs> Event;
```

Come suggerisce MSDN :

- Usa riferimenti deboli lunghi solo quando è necessario poiché lo stato dell'oggetto è imprevedibile dopo la finalizzazione.
- Evita di usare riferimenti deboli a oggetti piccoli perché il puntatore stesso potrebbe essere grande o più grande.
- Evitare l'uso di riferimenti deboli come soluzione automatica ai problemi di gestione della memoria. Invece, sviluppare una politica di caching efficace per la gestione degli oggetti dell'applicazione.

Leggi Garbage Collector in .Net online: https://riptutorial.com/it/csharp/topic/1287/garbage-collector-in--net

Capitolo 58: Generare numeri casuali in C

Sintassi

- Casuale()
- Casuale (int Seed)
- int Successivo ()
- int Next (int maxValue)
- int Successivo (int minValue, int maxValue)

Parametri

parametri	Dettagli
seme	Un valore per la generazione di numeri casuali. Se non impostato, il valore predefinito è determinato dall'ora corrente del sistema.
minValue	I numeri generati non saranno più piccoli di questo valore. Se non impostato, il valore predefinito è 0.
maxValue	I numeri generati saranno più piccoli di questo valore. Se non impostato, il valore predefinito è `Int32.MaxValue` .
valore di ritorno	Restituisce un numero con valore casuale.

Osservazioni

Il seme casuale generato dal sistema non è lo stesso in ogni altra corsa.

I semi generati nello stesso tempo potrebbero essere uguali.

Examples

Genera un int casuale

Questo esempio genera valori casuali tra 0 e 2147483647.

```
Random rnd = new Random();
int randomNumber = rnd.Next();
```

Genera un doppio casuale

Genera un numero casuale compreso tra 0 e 1.0. (escluso 1.0)

```
Random rnd = new Random();
var randomDouble = rnd.NextDouble();
```

Genera un int random in un dato range

Genera un numero casuale tra `minValue` e `maxValue - 1` .

```
Random rnd = new Random();
var randomBetween10And20 = rnd.Next(10, 20);
```

Generazione della stessa sequenza di numeri casuali più e più volte

Quando si creano istanze `Random` con lo stesso seme, verranno generati gli stessi numeri.

```
int seed = 5;
for (int i = 0; i < 2; i++)
{
   Console.WriteLine("Random instance " + i);
   Random rnd = new Random(seed);
   for (int j = 0; j < 5; j++)
   {
      Console.Write(rnd.Next());
      Console.Write(" ");
   }

   Console.WriteLine();
}
```

Produzione:

```
Random instance 0
726643700 610783965 564707973 1342984399 995276750
Random instance 1
726643700 610783965 564707973 1342984399 995276750
```

Crea più classi casuali con semi diversi contemporaneamente

Due classi casuali create contemporaneamente avranno lo stesso valore di inizializzazione.

Usando `System.Guid.NewGuid().GetHashCode()` può ottenere un seed diverso anche nello stesso tempo.

```
Random rnd1 = new Random();
Random rnd2 = new Random();
Console.WriteLine("First 5 random number in rnd1");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());
```

```
Console.WriteLine("First 5 random number in rnd2");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());

rnd1 = new Random(Guid.NewGuid().GetHashCode());
rnd2 = new Random(Guid.NewGuid().GetHashCode());
Console.WriteLine("First 5 random number in rnd1 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());
Console.WriteLine("First 5 random number in rnd2 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());
```

Un altro modo per ottenere semi diversi è usare un'altra istanza `Random` per recuperare i valori seme.

```
Random rndSeeds = new Random();
Random rnd1 = new Random(rndSeeds.Next());
Random rnd2 = new Random(rndSeeds.Next());
```

Ciò consente inoltre di controllare il risultato di tutte le istanze `Random` impostando solo il valore di `rndSeeds` per `rndSeeds` . Tutte le altre istanze saranno derivate deterministicamente dal valore di seme singolo.

Genera un personaggio casuale

Generare una lettera casuale tra `a` e `z` usando il sovraccarico `Next()` per un dato intervallo di numeri, quindi convertire l' `int` risultante in un `char`

```
Random rnd = new Random();
char randomChar = (char)rnd.Next('a','z');
//'a' and 'z' are interpreted as ints for parameters for Next()
```

Genera un numero che è una percentuale di un valore massimo

Un bisogno comune di numeri casuali per generare un numero che è `X%` di qualche valore massimo. questo può essere fatto trattando il risultato di `NextDouble()` come percentuale:

```
var rnd = new Random();
var maxValue = 5000;
var percentage = rnd.NextDouble();
var result = maxValue * percentage;
//suppose NextDouble() returns .65, result will hold 65% of 5000: 3250.
```

Leggi Generare numeri casuali in C # online: https://riptutorial.com/it/csharp/topic/1975/generare-numeri-casuali-in-c-sharp

Capitolo 59: Generatore di query Lambda generico

Osservazioni

La classe si chiama `ExpressionBuilder` . Ha tre proprietà:

```
 private static readonly MethodInfo ContainsMethod = typeof(string).GetMethod("Contains",
new[] { typeof(string) });
 private static readonly MethodInfo StartsWithMethod = typeof(string).GetMethod("StartsWith",
new[] { typeof(string) });
 private static readonly MethodInfo EndsWithMethod = typeof(string).GetMethod("EndsWith",
new[] { typeof(string) });
```

Un metodo pubblico `GetExpression` che restituisce l'espressione lambda e tre metodi privati:

- `Expression GetExpression<T>`
- `BinaryExpression GetExpression<T>`
- `ConstantExpression GetConstant`

Tutti i metodi sono spiegati in dettaglio negli esempi.

Examples

Classe QueryFilter

Questa classe contiene i valori dei filtri predicati.

```
public class QueryFilter
{
    public string PropertyName { get; set; }
    public string Value { get; set; }
    public Operator Operator { get; set; }

    // In the query {a => a.Name.Equals("Pedro")}
    // Property name to filter - propertyName = "Name"
    // Filter value - value = "Pedro"
    // Operation to perform - operation = enum Operator.Equals
    public QueryFilter(string propertyName, string value, Operator operatorValue)
    {
        PropertyName = propertyName;
        Value = value;
        Operator = operatorValue;
    }
}
```

Enum per contenere i valori delle operazioni:

```
 public enum Operator
```

```
{
    Contains,
    GreaterThan,
    GreaterThanOrEqual,
    LessThan,
    LessThanOrEqualTo,
    StartsWith,
    EndsWith,
    Equals,
    NotEqual
}
```

Metodo GetExpression

```
public static Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
{
    Expression exp = null;

    // Represents a named parameter expression. {parm => parm.Name.Equals()}, it is the param
part
    // To create a ParameterExpression need the type of the entity that the query is against
an a name
    // The type is possible to find with the generic T and the name is fixed parm
    ParameterExpression param = Expression.Parameter(typeof(T), "parm");

    // It is good parctice never trust in the client, so it is wise to validate.
    if (filters.Count == 0)
        return null;

    // The expression creation differ if there is one, two or more filters.
    if (filters.Count != 1)
    {
        if (filters.Count == 2)
            // It is result from direct call.
            // For simplicity sake the private overloads will be explained in another example.
            exp = GetExpression<T>(param, filters[0], filters[1]);
        else
        {
            // As there is no method for more than two filters,
            // I iterate through all the filters and put I in the query two at a time
            while (filters.Count > 0)
            {
                // Retreive the first two filters
                var f1 = filters[0];
                var f2 = filters[1];

                // To build a expression with a conditional AND operation that evaluates
                // the second operand only if the first operand evaluates to true.
                // It needed to use the BinaryExpression a Expression derived class
                // That has the AndAlso method that join two expression together
                exp = exp == null ? GetExpression<T>(param, filters[0], filters[1]) :
Expression.AndAlso(exp, GetExpression<T>(param, filters[0], filters[1]));

                // Remove the two just used filters, for the method in the next iteration
finds the next filters
                filters.Remove(f1);
                filters.Remove(f2);

                // If it is that last filter, add the last one and remove it
```

```
                if (filters.Count == 1)
                {
                    exp = Expression.AndAlso(exp, GetExpression<T>(param, filters[0]));
                    filters.RemoveAt(0);
                }
            }
        }
    }
    else
        // It is result from direct call.
        exp = GetExpression<T>(param, filters[0]);

           // converts the Expression into Lambda and retuns the query
    return Expression.Lambda<Func<T, bool>>(exp, param);
}
```

GetExpression Sovraccarico privato

Per un filtro:

Qui è dove viene creata la query, riceve un parametro di espressione e un filtro.

```
private static Expression GetExpression<T>(ParameterExpression param, QueryFilter queryFilter)
{
    // Represents accessing a field or property, so here we are accessing for example:
    // the property "Name" of the entity
    MemberExpression member = Expression.Property(param, queryFilter.PropertyName);

    //Represents an expression that has a constant value, so here we are accessing for
example:
    // the values of the Property "Name".
    // Also for clarity sake the GetConstant will be explained in another example.
    ConstantExpression constant = GetConstant(member.Type, queryFilter.Value);

    // With these two, now I can build the expression
    // every operator has it one way to call, so the switch will do.
    switch (queryFilter.Operator)
    {
        case Operator.Equals:
            return Expression.Equal(member, constant);

        case Operator.Contains:
            return Expression.Call(member, ContainsMethod, constant);

        case Operator.GreaterThan:
            return Expression.GreaterThan(member, constant);

        case Operator.GreaterThanOrEqual:
            return Expression.GreaterThanOrEqual(member, constant);

        case Operator.LessThan:
            return Expression.LessThan(member, constant);

        case Operator.LessThanOrEqualTo:
            return Expression.LessThanOrEqual(member, constant);

        case Operator.StartsWith:
```

```
            return Expression.Call(member, StartsWithMethod, constant);

        case Operator.EndsWith:
            return Expression.Call(member, EndsWithMethod, constant);
    }

    return null;
}
```

Per due filtri:

Restituisce l'istanza di BinaryExpresion invece della semplice espressione.

```
private static BinaryExpression GetExpression<T>(ParameterExpression param, QueryFilter
filter1, QueryFilter filter2)
{
    // Built two separated expression and join them after.
    Expression result1 = GetExpression<T>(param, filter1);
    Expression result2 = GetExpression<T>(param, filter2);
    return Expression.AndAlso(result1, result2);
}
```

Metodo ConstantExpression

`ConstantExpression` deve essere lo stesso tipo di `MemberExpression` . Il valore in questo esempio è una stringa, che viene convertita prima di creare l'istanza di `ConstantExpression` .

```
private static ConstantExpression GetConstant(Type type, string value)
{
    // Discover the type, convert it, and create ConstantExpression
    ConstantExpression constant = null;
    if (type == typeof(int))
    {
        int num;
        int.TryParse(value, out num);
        constant = Expression.Constant(num);
    }
    else if(type == typeof(string))
    {
        constant = Expression.Constant(value);
    }
    else if (type == typeof(DateTime))
    {
        DateTime date;
        DateTime.TryParse(value, out date);
        constant = Expression.Constant(date);
    }
    else if (type == typeof(bool))
    {
        bool flag;
        if (bool.TryParse(value, out flag))
        {
            flag = true;
        }
        constant = Expression.Constant(flag);
```

```
    }
    else if (type == typeof(decimal))
    {
        decimal number;
        decimal.TryParse(value, out number);
        constant = Expression.Constant(number);
    }
    return constant;
}
```

uso

Filtri di raccolta = nuova lista (); Filtro QueryFilter = new QueryFilter ("Nome", "Burger", Operator.StartsWith); filters.Add (filtro);

```
 Expression<Func<Food, bool>> query = ExpressionBuilder.GetExpression<Food>(filters);
```

In questo caso, si tratta di una query contro l'entità Cibo, che desidera trovare tutti gli alimenti che iniziano con "Burger" nel nome.

Produzione:

```
query = {parm => a.parm.StartsWith("Burger")}


Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
```

Leggi Generatore di query Lambda generico online:
https://riptutorial.com/it/csharp/topic/6721/generatore-di-query-lambda-generico

Capitolo 60: Generazione del codice T4

Sintassi

- **Sintassi T4**
- `<#@...#>` // Dichiarazione delle proprietà inclusi modelli, assiemi e spazi dei nomi e la lingua utilizzata dal modello
- `Plain Text` // Dichiarazione del testo che può essere ricollegato per i file generati
- `<#=...#>` // Dichiarare gli script
- `<#+...#>` // Dichiarare scriptlet
- `<#...#>` // Dichiarare blocchi di testo

Examples

Generazione del codice runtime

```
<#@ template language="C#" #> //Language of your project
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
```

Leggi Generazione del codice T4 online: https://riptutorial.com/it/csharp/topic/4824/generazione-del-codice-t4

Capitolo 61: Generics

Sintassi

- `public void SomeMethod <T> () { }`
- `public void SomeMethod<T, V>() { }`
- `public T SomeMethod<T>(IEnumerable<T> sequence) { ... }`
- `public void SomeMethod<T>() where T : new() { }`
- `public void SomeMethod<T, V>() where T : new() where V : struct { }`
- `public void SomeMethod<T>() where T: IDisposable { }`
- `public void SomeMethod<T>() where T: Foo { }`
- `public class MyClass<T> { public T Data {get; set; } }`

Parametri

Parametro (s)	Descrizione
TV	Digitare segnaposto per dichiarazioni generiche

Osservazioni

Generics in C # sono supportati fino al runtime: i tipi generici creati con C # conservano la loro semantica generica anche dopo essere stati compilati in CIL .

Ciò significa in effetti che, in C #, è possibile riflettere su tipi generici e vederli come sono stati dichiarati o controllare se un oggetto è un'istanza di un tipo generico, ad esempio. Ciò è in contrasto con la cancellazione del tipo , in cui le informazioni di tipo generico vengono rimosse durante la compilazione. È anche in contrasto con l'approccio del modello ai generici, in cui più tipi generici concreti diventano più tipi non generici in fase di runtime e tutti i metadati necessari per un'ulteriore definizione delle definizioni del tipo generico originale vengono persi.

Prestare attenzione, tuttavia, quando si riflette su tipi generici: i nomi dei tipi generici verranno alterati nella compilazione, sostituendo le parentesi angolate ei nomi dei parametri di tipo con un apice seguito dal numero di parametri di tipo generico. Pertanto, un `Dictionary<TKey, Tvalue>` sarà tradotto nel `Dictionary`2` .

Examples

Tipo Parametri (Classi)

Dichiarazione:

```
class MyGenericClass<T1, T2, T3, ...>
{
    // Do something with the type parameters.
```

```
}
```

Inizializzazione:

```
var x = new MyGenericClass<int, char, bool>();
```

Utilizzo (come il tipo di un parametro):

```
void AnotherMethod(MyGenericClass<float, byte, char> arg) { ... }
```

Tipo Parametri (metodi)

Dichiarazione:

```
void MyGenericMethod<T1, T2, T3>(T1 a, T2 b, T3 c)
{
    // Do something with the type parameters.
}
```

Invocazione:

Non è necessario fornire argomentazioni di tipo a un metodo genrico, poiché il compilatore può implicitamente inferire il tipo.

```
int x =10;
int y =20;
string z = "test";
MyGenericMethod(x,y,z);
```

Tuttavia, se c'è un'ambiguità, i metodi generici devono essere chiamati con tipo arguemnts come

```
MyGenericMethod<int, int, string>(x,y,z);
```

Tipo Parametri (interfacce)

Dichiarazione:

```
interface IMyGenericInterface<T1, T2, T3, ...> { ... }
```

Uso (in eredità):

```
class ClassA<T1, T2, T3> : IMyGenericInterface<T1, T2, T3> { ... }

class ClassB<T1, T2> : IMyGenericInterface<T1, T2, int> { ... }

class ClassC<T1> : IMyGenericInterface<T1, char, int> { ... }

class ClassD : IMyGenericInterface<bool, char, int> { ... }
```

Utilizzo (come il tipo di un parametro):

```
void SomeMethod(IMyGenericInterface<int, char, bool> arg) { ... }
```

Inferenza implicita del tipo (metodi)

Quando si passano argomenti formali a un metodo generico, argomenti di tipo generico rilevanti possono essere normalmente dedotti implicitamente. Se è possibile dedurre tutti i tipi generici, specificarli nella sintassi è facoltativo.

Si consideri il seguente metodo generico. Ha un parametro formale e un parametro di tipo generico. Esiste una relazione molto evidente tra di essi: il tipo passato come argomento al parametro di tipo generico deve essere uguale al tipo di tempo di compilazione dell'argomento passato al parametro formale.

```
void M<T>(T obj)
{
}
```

Queste due chiamate sono equivalenti:

```
M<object>(new object());
M(new object());
```

Queste due chiamate sono anche equivalenti:

```
M<string>("");
M("");
```

E così sono queste tre chiamate:

```
M<object>("");
M((object) "");
M("" as object);
```

Si noti che se non è possibile dedurre almeno un argomento di tipo, è necessario specificarli tutti.

Si consideri il seguente metodo generico. Il primo argomento di tipo generico è lo stesso del tipo dell'argomento formale. Ma non esiste una tale relazione per il secondo argomento di tipo generico. Pertanto, il compilatore non ha modo di inferire il secondo argomento di tipo generico in qualsiasi chiamata a questo metodo.

```
void X<T1, T2>(T1 obj)
{
}
```

Questo non funziona più:

```
X("");
```

Anche questo non funziona, perché il compilatore non è sicuro se stiamo specificando il primo o il secondo parametro generico (entrambi sarebbero validi come `object`):

```
X<object>("");
```

Siamo tenuti a digitare entrambi, in questo modo:

```
X<string, object>("");
```

Vincoli di tipo (classi e interfacce)

I vincoli di tipo sono in grado di forzare un parametro di tipo per implementare una determinata interfaccia o classe.

```
interface IType;
interface IAnotherType;

// T must be a subtype of IType
interface IGeneric<T>
    where T : IType
{
}

// T must be a subtype of IType
class Generic<T>
    where T : IType
{
}

class NonGeneric
{
    // T must be a subtype of IType
    public void DoSomething<T>(T arg)
        where T : IType
    {
    }
}

// Valid definitions and expressions:
class Type : IType { }
class Sub : IGeneric<Type> { }
class Sub : Generic<Type> { }
new NonGeneric().DoSomething(new Type());

// Invalid definitions and expressions:
class AnotherType : IAnotherType { }
class Sub : IGeneric<AnotherType> { }
class Sub : Generic<AnotherType> { }
new NonGeneric().DoSomething(new AnotherType());
```

Sintassi per più vincoli:

```
class Generic<T, T1>
    where T : IType
    where T1 : Base, new()
{
}
```

I vincoli di tipo funzionano allo stesso modo dell'ereditarietà, in quanto è possibile specificare più interfacce come vincoli sul tipo generico, ma solo una classe:

```
class A { /* ... */ }
class B { /* ... */ }

interface I1 { }
interface I2 { }

class Generic<T>
    where T : A, I1, I2
{
}

class Generic2<T>
    where T : A, B //Compilation error
{
}
```

Un'altra regola è che la classe deve essere aggiunta come primo vincolo e quindi le interfacce:

```
class Generic<T>
    where T : A, I1
{
}

class Generic2<T>
    where T : I1, A //Compilation error
{
}
```

Tutti i vincoli dichiarati devono essere soddisfatti simultaneamente affinché una determinata istanza generica funzioni. Non c'è modo di specificare due o più insiemi alternativi di vincoli.

Vincoli di tipo (classe e struct)

È possibile specificare se l'argomento type debba essere un tipo di riferimento o un tipo di valore utilizzando la rispettiva `class` vincoli o `struct` . Se questi vincoli vengono utilizzati, *devono* essere definiti *prima di* poter elencare *tutti gli* altri vincoli (ad esempio un genitore o un `new()`).

```
// TRef must be a reference type, the use of Int32, Single, etc. is invalid.
// Interfaces are valid, as they are reference types
class AcceptsRefType<TRef>
    where TRef : class
{
    // TStruct must be a value type.
    public void AcceptStruct<TStruct>()
        where TStruct : struct
    {
```

```
    }

    // If multiple constraints are used along with class/struct
    // then the class or struct constraint MUST be specified first
    public void Foo<TComparableClass>()
        where TComparableClass : class, IComparable
    {
    }
}
```

Vincoli di tipo (nuova parola chiave)

Utilizzando il `new()` vincolo `new()` , è possibile applicare i parametri del tipo per definire un costruttore vuoto (predefinito).

```
class Foo
{
    public Foo () { }
}

class Bar
{
    public Bar (string s) { ... }
}

class Factory<T>
    where T : new()
{
    public T Create()
    {
        return new T();
    }
}

Foo f = new Factory<Foo>().Create(); // Valid.
Bar b = new Factory<Bar>().Create(); // Invalid, Bar does not define a default/empty
constructor.
```

La seconda chiamata a `Create()` restituirà l'errore di compilazione con il seguente messaggio:

> 'Bar' deve essere un tipo non astratto con un costruttore pubblico senza parametri per utilizzarlo come parametro 'T' nel tipo generico o nel metodo 'Factory'

Non esiste alcun vincolo per un costruttore con parametri, sono supportati solo costruttori senza parametri.

Inferenza di tipo (classi)

Gli sviluppatori possono essere colti dal fatto che l'inferenza di tipo *non funziona* per i costruttori:

```
class Tuple<T1,T2>
{
   public Tuple(T1 value1, T2 value2)
   {
   }
```

```
}

var x = new Tuple(2, "two");              // This WON'T work...
var y = new Tuple<int, string>(2, "two"); // even though the explicit form will.
```

Il primo modo di creare un'istanza senza specificare esplicitamente i parametri di tipo causerà errori di compilazione che direbbero:

> L'utilizzo del tipo generico 'Tuple <T1, T2>' richiede due argomenti di tipo

Una soluzione alternativa è aggiungere un metodo helper in una classe statica:

```
static class Tuple
{
    public static Tuple<T1, T2> Create<T1, T2>(T1 value1, T2 value2)
    {
         return new Tuple<T1, T2>(value1, value2);
    }
}

var x = Tuple.Create(2, "two");  // This WILL work...
```

Riflettendo sui parametri del tipo

L'operatore `typeof` funziona sui parametri del tipo.

```
class NameGetter<T>
{
    public string GetTypeName()
    {
        return typeof(T).Name;
    }
}
```

Parametri di tipo esplicito

Esistono diversi casi in cui è necessario specificare esplicitamente i parametri del tipo per un metodo generico. In entrambi i casi seguenti, il compilatore non è in grado di dedurre tutti i parametri del tipo dai parametri del metodo specificato.

Un caso è quando non ci sono parametri:

```
public void SomeMethod<T, V>()
{
   // No code for simplicity
}

SomeMethod(); // doesn't compile
SomeMethod<int, bool>(); // compiles
```

Il secondo caso è quando uno (o più) dei parametri del tipo non fa parte dei parametri del metodo:

```
public K SomeMethod<K, V>(V input)
{
    return default(K);
}

int num1 = SomeMethod(3); // doesn't compile
int num2 = SomeMethod<int>("3"); // doesn't compile
int num3 = SomeMethod<int, string>("3"); // compiles.
```

Utilizzo del metodo generico con un'interfaccia come tipo di vincolo.

Questo è un esempio di come usare il tipo generico TFood all'interno del metodo Eat sulla classe Animal

```
public interface IFood
{
    void EatenBy(Animal animal);
}

public class Grass: IFood
{
    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Grass was eaten by: {0}", animal.Name);
    }
}

public class Animal
{
    public string Name { get; set; }

    public void Eat<TFood>(TFood food)
        where TFood : IFood
    {
        food.EatenBy(this);
    }
}

public class Carnivore : Animal
{
    public Carnivore()
    {
        Name = "Carnivore";
    }
}

public class Herbivore : Animal, IFood
{
    public Herbivore()
    {
        Name = "Herbivore";
    }

    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Herbivore was eaten by: {0}", animal.Name);
    }
}
```

Puoi chiamare il metodo Eat in questo modo:

```
var grass = new Grass();
var sheep = new Herbivore();
var lion = new Carnivore();

sheep.Eat(grass);
//Output: Grass was eaten by: Herbivore

lion.Eat(sheep);
//Output: Herbivore was eaten by: Carnivore
```

In questo caso, se provi a chiamare:

```
sheep.Eat(lion);
```

Non sarà possibile perché l'oggetto leone non implementa l'interfaccia IFood. Tentare di effettuare la chiamata sopra genererà un errore del compilatore: "Il tipo 'Carnivore' non può essere utilizzato come parametro di tipo 'TFood' nel tipo generico o nel metodo 'Animal.Eat (TFood)'. Non c'è conversione implicita di riferimento da ' Carnivore 'a' IFood '. "

covarianza

Quando è un `IEnumerable<T>` un sottotipo di un `IEnumerable<T1>` diverso? Quando `T` è un sottotipo di `T1` . `IEnumerable` è *covariante* nel suo parametro `T` , il che significa che la relazione del sottotipo di `IEnumerable` va nella *stessa direzione* di `T`

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IEnumerable<Dog> dogs = Enumerable.Empty<Dog>();
IEnumerable<Animal> animals = dogs;  // IEnumerable<Dog> is a subtype of IEnumerable<Animal>
// dogs = animals;  // Compilation error - IEnumerable<Animal> is not a subtype of
IEnumerable<Dog>
```

Un'istanza di un tipo generico covariante con un determinato parametro di tipo è implicitamente convertibile nello stesso tipo generico con un parametro di tipo meno derivato.

Questa relazione vale perché `IEnumerable` *produce* `T` s ma non li consuma. Un oggetto che produce `Dog` s può essere usato come se producesse `Animal` s.

I parametri di tipo Covariant vengono dichiarati utilizzando la parola chiave `out` , poiché il parametro deve essere utilizzato solo come *output* .

```
interface IEnumerable<out T> { /* ... */ }
```

Un parametro di tipo dichiarato come covariante potrebbe non apparire come input.

```
interface Bad<out T>
{
    void SetT(T t);  // type error
```

```
}
```

Ecco un esempio completo:

```
using NUnit.Framework;

namespace ToyStore
{
   enum Taste { Bitter, Sweet };

   interface IWidget
   {
      int Weight { get; }
   }

   interface IFactory<out TWidget>
       where TWidget : IWidget
   {
      TWidget Create();
   }

   class Toy : IWidget
   {
      public int Weight { get; set; }
      public Taste Taste { get; set; }
   }

   class ToyFactory : IFactory<Toy>
   {
      public const int StandardWeight = 100;
      public const Taste StandardTaste = Taste.Sweet;

      public Toy Create() { return new Toy { Weight = StandardWeight, Taste = StandardTaste };
}
   }

   [TestFixture]
   public class GivenAToyFactory
   {
      [Test]
      public static void WhenUsingToyFactoryToMakeWidgets()
      {
         var toyFactory = new ToyFactory();

         //// Without out keyword, note the verbose explicit cast:
         // IFactory<IWidget> rustBeltFactory = (IFactory<IWidget>)toyFactory;

         // covariance: concrete being assigned to abstract (shiny and new)
         IFactory<IWidget> widgetFactory = toyFactory;
         IWidget anotherToy = widgetFactory.Create();
         Assert.That(anotherToy.Weight, Is.EqualTo(ToyFactory.StandardWeight)); // abstract
contract
         Assert.That(((Toy)anotherToy).Taste, Is.EqualTo(ToyFactory.StandardTaste)); //
concrete contract
      }
   }
}
```

controvarianza

Quando è un `IComparer<T>` un sottotipo di un altro `IComparer<T1>` ? Quando `T1` è un sottotipo di `T` `IComparer` è *controvariante* nel suo parametro `T` , il che significa che la relazione del sottotipo di `IComparer` va nella *direzione opposta a* quella di `T`

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IComparer<Animal> animalComparer = /* ... */;
IComparer<Dog> dogComparer = animalComparer;  // IComparer<Animal> is a subtype of
IComparer<Dog>
// animalComparer = dogComparer;  // Compilation error - IComparer<Dog> is not a subtype of
IComparer<Animal>
```

Un'istanza di un tipo generico controvariante con un determinato parametro di tipo è implicitamente convertibile nello stesso tipo generico con un parametro di tipo più derivato.

Questa relazione vale perché `IComparer` *consuma* `T` s ma non li produce. Un oggetto che può confrontare due `Animal` può essere usato per confrontare due `Dog` .

Parametri di tipo controvarianti vengono dichiarati utilizzando la `in` parola, poiché il parametro deve essere utilizzato solo come *ingresso.*

```
interface IComparer<in T> { /* ... */ }
```

Un parametro di tipo dichiarato come controvariante potrebbe non apparire come output.

```
interface Bad<in T>
{
    T GetT();  // type error
}
```

invarianza

`IList<T>` non è mai un sottotipo di un diverso `IList<T1>` . `IList` è *invariante* nel suo parametro di tipo.

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IList<Dog> dogs = new List<Dog>();
IList<Animal> animals = dogs;  // type error
```

Non esiste una relazione di sottotipo per gli elenchi poiché è possibile inserire valori in un elenco *e* ricavare valori da un elenco.

Se `IList` fosse covariante, saresti in grado di aggiungere elementi del *sottotipo sbagliato* a una data lista.

```
IList<Animal> animals = new List<Dog>();  // supposing this were allowed...
animals.Add(new Giraffe());  // ... then this would also be allowed, which is bad!
```

Se `IList` fosse controverso, saresti in grado di estrarre i valori del sottotipo errato da una lista data.

```
IList<Dog> dogs = new List<Animal> { new Dog(), new Giraffe() };  // if this were allowed...
Dog dog = dogs[1];  // ... then this would be allowed, which is bad!
```

I parametri di tipo invariante sono dichiarati omettendo sia le parole chiave `in` e `out` .

```
interface IList<T> { /* ... */ }
```

Interfacce varianti

Le interfacce possono avere parametri di tipo variante.

```
interface IEnumerable<out T>
{
    // ...
}
interface IComparer<in T>
{
    // ...
}
```

ma le classi e le strutture non possono

```
class BadClass<in T1, out T2>  // not allowed
{
}

struct BadStruct<in T1, out T2>  // not allowed
{
}
```

né dichiarazioni di metodi generici

```
class MyClass
{
    public T Bad<out T, in T1>(T1 t1)  // not allowed
    {
        // ...
    }
}
```

L'esempio seguente mostra più dichiarazioni di varianza sulla stessa interfaccia

```
interface IFoo<in T1, out T2, T3>
//  T1 : Contravariant type
//  T2 : Covariant type
//  T3 : Invariant type
{
    // ...
}
```

```
IFoo<Animal, Dog, int> foo1 = /* ... */;
IFoo<Dog, Animal, int> foo2 = foo1;
// IFoo<Animal, Dog, int> is a subtype of IFoo<Dog, Animal, int>
```

Delegati varianti

I delegati possono avere parametri di tipo variante.

```
delegate void Action<in T>(T t);    // T is an input
delegate T Func<out T>();           // T is an output
delegate T2 Func<in T1, out T2>();  // T1 is an input, T2 is an output
```

Ciò deriva dal Principio di sostituzione di Liskov , che stabilisce (tra le altre cose) che un metodo D può essere considerato più derivato di un metodo B se:

- D ha un tipo di ritorno uguale o più derivato di B
- D ha i tipi di parametro corrispondenti uguali o più generali di B

Pertanto i seguenti compiti sono tutti di tipo sicuro:

```
Func<object, string> original = SomeMethod;
Func<object, object> d1 = original;
Func<string, string> d2 = original;
Func<string, object> d3 = original;
```

Tipi varianti come parametri e valori di ritorno

Se un tipo covariante viene visualizzato come output, il tipo contenente è covariante. Produrre un produttore di `T` s è come produrre `T` s.

```
interface IReturnCovariant<out T>
{
    IEnumerable<T> GetTs();
}
```

Se un tipo controvariante appare come output, il tipo contenente è controverso. Produrre un consumatore di `T` s è come consumare `T` s.

```
interface IReturnContravariant<in T>
{
    IComparer<T> GetTComparer();
}
```

Se un tipo covariante appare come un input, il tipo contenente è controverso. Consumare un produttore di `T` s è come consumare `T` s.

```
interface IAcceptCovariant<in T>
{
    void ProcessTs(IEnumerable<T> ts);
}
```

Se un tipo controvariante appare come input, il tipo contenente è covariante. Consumare un consumatore di `T` s è come produrre `T` s.

```
interface IAcceptContravariant<out T>
{
    void CompareTs(IComparer<T> tComparer);
}
```

Controllo dell'uguaglianza dei valori generici.

Se la logica di classe generica o metodo richiede il controllo parità di valore aventi tipo generico, utilizzare `EqualityComparer<TType>.Default` proprietà :

```
public void Foo<TBar>(TBar arg1, TBar arg2)
{
    var comparer = EqualityComparer<TBar>.Default;
    if (comparer.Equals(arg1,arg2)
    {
        ...
    }
}
```

Questo approccio è meglio che semplicemente chiamando `Object.Equals()` metodo, perché i controlli di implementazione predefinita di confronto, se `TBar` tipo implementa `IEquatale<TBar>` Interfaccia e se sì, le chiamate `IEquatable<TBar>.Equals(TBar other)` metodo. Ciò consente di evitare il boxing / unboxing dei tipi di valore.

Tipo generico casting

```
    /// <summary>
    /// Converts a data type to another data type.
    /// </summary>
    public static class Cast
    {
        /// <summary>
        /// Converts input to Type of default value or given as typeparam T
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="input">Input that need to be converted to specified type</param>
        /// <param name="defaultValue">defaultValue will be returned in case of value is null
or any exception occures</param>
        /// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
        public static T To<T>(object input, T defaultValue)
        {
            var result = defaultValue;
            try
            {
                if (input == null || input == DBNull.Value) return result;
                if (typeof (T).IsEnum)
                {
                    result = (T) Enum.ToObject(typeof (T), To(input,
Convert.ToInt32(defaultValue)));
                }
```

```
                else
                {
                    result = (T) Convert.ChangeType(input, typeof (T));
                }
            }
            catch (Exception ex)
            {
                Tracer.Current.LogException(ex);
            }

            return result;
        }

        /// <summary>
        /// Converts input to Type of typeparam T
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="input">Input that need to be converted to specified type</param>
        /// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
        public static T To<T>(object input)
        {
            return To(input, default(T));
        }



    }
```

usi:

```
std.Name = Cast.To<string>(drConnection["Name"]);
std.Age = Cast.To<int>(drConnection["Age"]);
std.IsPassed = Cast.To<bool>(drConnection["IsPassed"]);


// Casting type using default value
//Following both ways are correct
// Way 1 (In following style input is converted into type of default value)
std.Name = Cast.To(drConnection["Name"], "");
std.Marks = Cast.To(drConnection["Marks"], 0);
// Way 2
std.Name = Cast.To<string>(drConnection["Name"], "");
std.Marks = Cast.To<int>(drConnection["Marks"], 0);
```

Lettore di configurazione con casting di tipo generico

```
    /// <summary>
    /// Read configuration values from app.config and convert to specified types
    /// </summary>
    public static class ConfigurationReader
    {
        /// <summary>
        /// Get value from AppSettings by key, convert to Type of default value or typeparam T
and return
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
```

```
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="strKey">key to find value from AppSettings</param>
        /// <param name="defaultValue">defaultValue will be returned in case of value is null
or any exception occures</param>
        /// <returns>AppSettings value against key is returned in Type of default value or
given as typeparam T</returns>
        public static T GetConfigKeyValue<T>(string strKey, T defaultValue)
        {
            var result = defaultValue;
            try
            {
                if (ConfigurationManager.AppSettings[strKey] != null)
                    result = (T)Convert.ChangeType(ConfigurationManager.AppSettings[strKey],
typeof(T));
            }
            catch (Exception ex)
            {
                Tracer.Current.LogException(ex);
            }

            return result;
        }
        /// <summary>
        /// Get value from AppSettings by key, convert to Type of default value or typeparam T
and return
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it
could be any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="strKey">key to find value from AppSettings</param>
        /// <returns>AppSettings value against key is returned in Type given as typeparam
T</returns>
        public static T GetConfigKeyValue<T>(string strKey)
        {
            return GetConfigKeyValue(strKey, default(T));
        }

    }
```

usi:

```
var timeOut = ConfigurationReader.GetConfigKeyValue("RequestTimeout", 2000);
var url = ConfigurationReader.GetConfigKeyValue("URL", "www.someurl.com");
var enabled = ConfigurationReader.GetConfigKeyValue("IsEnabled", false);
```

Leggi Generics online: https://riptutorial.com/it/csharp/topic/27/generics

Capitolo 62: Gestione di FormatException durante la conversione di stringhe in altri tipi

Examples

Conversione da stringa a intero

Sono disponibili vari metodi per convertire esplicitamente una `string` in un `integer` , ad esempio:

1. `Convert.ToInt16();`
2. `Convert.ToInt32();`
3. `Convert.ToInt64();`
4. `int.Parse();`

Ma tutti questi metodi generano un `FormatException` , se la stringa di input contiene caratteri non numerici. Per questo, abbiamo bisogno di scrivere un'ulteriore gestione delle eccezioni (`try..catch`) per trattarli in questi casi.

Spiegazione con esempi:

Quindi, lascia che i nostri input siano:

```
string inputString = "10.2";
```

Esempio 1: `Convert.ToInt32()`

```
int convertedInt = Convert.ToInt32(inputString); // Failed to Convert
// Throws an Exception "Input string was not in a correct format."
```

***Nota: Io** stesso vale per gli altri metodi citati, vale a dire: `Convert.ToInt16();` e `Convert.ToInt64();`*

Esempio 2: `int.Parse()`

```
int convertedInt = int.Parse(inputString); // Same result "Input string was not in a correct
format.
```

Come aggiriamo questo?

Come detto in precedenza, per gestire le eccezioni di solito abbiamo bisogno di un `try..catch` come mostrato di seguito:

```
try
{
```

```
    string inputString = "10.2";
    int convertedInt = int.Parse(inputString);
}
catch (Exception Ex)
{
    //Display some message, that the conversion has failed.
}
```

Ma usare il `try..catch` ovunque non sarà una buona pratica, e potrebbero esserci alcuni scenari in cui vogliamo dare `0` se l'input è sbagliato, *(Se seguiamo il metodo sopra, dobbiamo assegnare `0` a `convertedInt` da blocco di cattura).* Per gestire tali scenari possiamo utilizzare un metodo speciale chiamato `.TryParse()` .

Il metodo `.TryParse()` con una gestione interna delle eccezioni, che fornisce l'output al parametro `out` e restituisce un valore booleano che indica lo stato della conversione *(`true` se la conversione ha avuto esito positivo, `false` se non è riuscita).* Sulla base del valore di ritorno possiamo determinare lo stato della conversione. Vediamo un esempio:

Uso 1: memorizza il valore di ritorno in una variabile booleana

```
 int convertedInt; // Be the required integer
 bool isSuccessConversion = int.TryParse(inputString, out convertedInt);
```

Possiamo controllare la variabile `isSuccessConversion` dopo l'esecuzione per controllare lo stato della conversione. Se è falso, il valore di `convertedInt` sarà `0` *(non è necessario controllare il valore restituito se si desidera `0` per errore di conversione).*

Uso 2: controllare il valore restituito con `if`

```
if (int.TryParse(inputString, out convertedInt))
{
    // convertedInt will have the converted value
    // Proceed with that
}
else
{
 // Display an error message
}
```

Uso 3: senza controllare il valore di ritorno è possibile utilizzare quanto segue, se non si cura del valore restituito *(convertito o meno, `0` sarà ok)*

```
int.TryParse(inputString, out convertedInt);
// use the value of convertedInt
// But it will be 0 if not converted
```

Leggi Gestione di FormatException durante la conversione di stringhe in altri tipi online: https://riptutorial.com/it/csharp/topic/2886/gestione-di-formatexception-durante-la-conversione-di-stringhe-in-altri-tipi

Capitolo 63: Gestore dell'autenticazione C

Examples

Gestore di autenticazione

```
 public class AuthenticationHandler : DelegatingHandler
    {
        /// <summary>
        /// Holds request's header name which will contains token.
        /// </summary>
        private const string securityToken = "__RequestAuthToken";

        /// <summary>
        /// Default overridden method which performs authentication.
        /// </summary>
        /// <param name="request">Http request message.</param>
        /// <param name="cancellationToken">Cancellation token.</param>
        /// <returns>Returns http response message of type <see cref="HttpResponseMessage"/>
class asynchronously.</returns>
        protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
CancellationToken cancellationToken)
        {
            if (request.Headers.Contains(securityToken))
            {
                bool authorized = Authorize(request);
                if (!authorized)
                {
                    return ApiHttpUtility.FromResult(request, false,
HttpStatusCode.Unauthorized, MessageTypes.Error, Resource.UnAuthenticatedUser);
                }
            }
            else
            {
                return ApiHttpUtility.FromResult(request, false, HttpStatusCode.BadRequest,
MessageTypes.Error, Resource.UnAuthenticatedUser);
            }

            return base.SendAsync(request, cancellationToken);
        }

        /// <summary>
        /// Authorize user by validating token.
        /// </summary>
        /// <param name="requestMessage">Authorization context.</param>
        /// <returns>Returns a value indicating whether current request is authenticated or
not.</returns>
        private bool Authorize(HttpRequestMessage requestMessage)
        {
            try
            {
                HttpRequest request = HttpContext.Current.Request;
                string token = request.Headers[securityToken];
                return SecurityUtility.IsTokenValid(token, request.UserAgent,
HttpContext.Current.Server.MapPath("~/Content/"), requestMessage);
            }
            catch (Exception)
```

```
            {
                return false;
            }
        }
    }
```

Leggi Gestore dell'autenticazione C # online: https://riptutorial.com/it/csharp/topic/5430/gestore-dell-autenticazione-c-sharp

Capitolo 64: getto

Osservazioni

La trasmissione non è la stessa cosa di *Conversione* . È possibile convertire il valore di stringa `"-1"` in un valore intero (`-1`), ma ciò deve essere eseguito tramite metodi di libreria come `Convert.ToInt32()` o `Int32.Parse()` . Non può essere fatto usando direttamente la sintassi del cast.

Examples

Trasmetti un oggetto a un tipo di base

Date le seguenti definizioni:

```
public interface IMyInterface1
{
    string GetName();
}

public interface IMyInterface2
{
    string GetName();
}

public class MyClass : IMyInterface1, IMyInterface2
{
    string IMyInterface1.GetName()
    {
        return "IMyInterface1";
    }

    string IMyInterface2.GetName()
    {
        return "IMyInterface2";
    }
}
```

Trasmettere un oggetto ad un esempio di tipo base:

```
    MyClass obj = new MyClass();

    IMyInterface1 myClass1 = (IMyInterface1)obj;
    IMyInterface2 myClass2 = (IMyInterface2)obj;

    Console.WriteLine("I am : {0}", myClass1.GetName());
    Console.WriteLine("I am : {0}", myClass2.GetName());

    // Outputs :
    // I am : IMyInterface1
    // I am : IMyInterface2
```

Casting esplicito

Se si sa che un valore è di un tipo specifico, è possibile eseguirlo esplicitamente su quel tipo per poterlo utilizzare in un contesto in cui è necessario quel tipo.

```
object value = -1;
int number = (int) value;
Console.WriteLine(Math.Abs(number));
```

Se provassimo a passare il `value` direttamente a `Math.Abs()` , avremmo ottenuto un'eccezione in fase di compilazione perché `Math.Abs()` non ha un sovraccarico che accetta un `object` come parametro.

Se non è possibile eseguire il cast di un `value` su un `int` , la seconda riga in questo esempio genera una `InvalidCastException`

Safe Explicit Casting (operatore `as`)

Se non sei sicuro che un valore sia del tipo che pensi di essere, puoi lanciarlo in sicurezza usando l'operatore `as` . Se il valore non è di quel tipo, il valore risultante sarà `null` .

```
object value = "-1";
int? number = value as int?;
if(number != null)
{
    Console.WriteLine(Math.Abs(number.Value));
}
```

Si noti che `null` valori `null` non hanno alcun tipo, quindi la parola chiave `as` renderà sicuramente `null` quando si esegue il cast di qualsiasi valore `null` .

Casting implicito

Un valore verrà automaticamente convertito nel tipo appropriato se il compilatore sa che può sempre essere convertito in quel tipo.

```
int number = -1;
object value = number;
Console.WriteLine(value);
```

In questo esempio, non è stato necessario utilizzare la tipica sintassi di trasmissione esplicita perché il compilatore sa che tutti gli `int` possono essere trasmessi agli `object` s. In effetti, potremmo evitare di creare variabili e passare `-1` direttamente come argomento di `Console.WriteLine()` che si aspetta un `object` .

```
Console.WriteLine(-1);
```

Controllo della compatibilità senza casting

Se è necessario sapere se il tipo di un valore estende o implementa un determinato tipo, ma non si desidera eseguire effettivamente il cast come quel tipo, è possibile utilizzare l'operatore `is` .

```
if(value is int)
{
   Console.WriteLine(value + "is an int");
}
```

Conversioni numeriche esplicite

Gli operatori di casting espliciti possono essere utilizzati per eseguire conversioni di tipi numerici, anche se non si estendono o si implementano a vicenda.

```
double value = -1.1;
int number = (int) value;
```

Si noti che nei casi in cui il tipo di destinazione ha meno precisione rispetto al tipo originale, la precisione andrà persa. Ad esempio, `-1.1` come valore doppio nell'esempio precedente diventa `-1` come valore intero.

Inoltre, le conversioni numeriche si basano sui tipi in fase di compilazione, quindi non funzioneranno se i tipi numerici sono stati "incapsulati" in oggetti.

```
object value = -1.1;
int number = (int) value; // throws InvalidCastException
```

Operatori di conversione

In C #, i tipi possono definire *operatori di conversione* personalizzati, che consentono di convertire i valori in e da altri tipi utilizzando cast espliciti o impliciti. Ad esempio, considera una classe che intende rappresentare un'espressione JavaScript:

```
public class JsExpression
{
    private readonly string expression;
    public JsExpression(string rawExpression)
    {
        this.expression = rawExpression;
    }
    public override string ToString()
    {
        return this.expression;
    }
    public JsExpression IsEqualTo(JsExpression other)
    {
        return new JsExpression("(" + this + " == " + other + ")");
    }
}
```

Se volessimo creare un JsExpression che rappresenti un confronto tra due valori JavaScript, potremmo fare qualcosa del genere:

```
JsExpression intExpression = new JsExpression("-1");
JsExpression doubleExpression = new JsExpression("-1.0");
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

Ma possiamo aggiungere alcuni *operatori di conversione espliciti* a `JsExpression` , per consentire una conversione semplice quando si utilizza il cast esplicito.

```
public static explicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static explicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = (JsExpression)(-1);
JsExpression doubleExpression = (JsExpression)(-1.0);
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

Oppure, potremmo cambiare questi operatori in *implicito* per rendere la sintassi molto più semplice.

```
public static implicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static implicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = -1;
Console.WriteLine(intExpression.IsEqualTo(-1.0)); // (-1 == -1.0)
```

Operazioni di fusione LINQ

Supponiamo di avere tipi come il seguente:

```
interface IThing {  }
class Thing : IThing {  }
```

LINQ consente di creare una proiezione che modifica il tipo generico in fase di compilazione di un oggetto `IEnumerable<>` tramite i metodi di estensione `Enumerable.Cast<>()` ed `Enumerable.OfType<>()` .

```
IEnumerable<IThing> things = new IThing[] {new Thing()};
IEnumerable<Thing> things2 = things.Cast<Thing>();
IEnumerable<Thing> things3 = things.OfType<Thing>();
```

Quando viene valutato `things2` , il metodo `Cast<>()` proverà a `things2` tutti i valori nelle `things` in `Thing` s. Se incontra un valore che non può essere lanciato, verrà lanciata una

`InvalidCastException` .

Quando viene valutato `things3` , il `OfType<>()` farà lo stesso, eccetto che se incontra un valore che non può essere lanciato, semplicemente ometterà quel valore anziché lanciare un'eccezione.

A causa del tipo generico di questi metodi, non possono richiamare operatori di conversione o eseguire conversioni numeriche.

```
double[] doubles = new[]{1,2,3}.Cast<double>().ToArray(); // Throws InvalidCastException
```

Puoi semplicemente eseguire un cast all'interno di `.Select()` come soluzione alternativa:

```
double[] doubles = new[]{1,2,3}.Select(i => (double)i).ToArray();
```

Leggi getto online: https://riptutorial.com/it/csharp/topic/2690/getto

Capitolo 65: guid

introduzione

GUID (o UUID) è l'acronimo di "Globally Unique Identifier" (o "Universalally Identifier"). È un numero intero a 128 bit utilizzato per identificare le risorse.

Osservazioni

`Guid` s sono *identificatori univoci globali,* noto anche come *UUID* s', *uuid.*

Sono valori pseudocasuali a 128 bit. Ci sono così tanti validi `Guid` (circa 10 ^ 18 `Guid` per ogni cellula di ogni popolo sulla Terra) che se sono generati da un buon algoritmo pseudocasuale, possono essere considerati unici nell'intero universo con tutti i mezzi pratici.

`Guid` vengono spesso utilizzate come chiavi primarie nei database. Il loro vantaggio è che non è necessario chiamare il database per ottenere un nuovo ID che è (quasi) garantito come univoco.

Examples

Ottenere la rappresentazione stringa di un Guid

Una rappresentazione in formato stringa di una guida può essere ottenuta utilizzando il metodo `ToString` incorporato

```
string myGuidString = myGuid.ToString();
```

A seconda delle esigenze, è inoltre possibile formattare il Guid, aggiungendo un argomento del tipo di formato alla chiamata `ToString` .

```
var guid = new Guid("7febf16f-651b-43b0-a5e3-0da8da49e90d");

// None          "7febf16f651b43b0a5e30da8da49e90d"
Console.WriteLine(guid.ToString("N"));

// Hyphens       "7febf16f-651b-43b0-a5e3-0da8da49e90d"
Console.WriteLine(guid.ToString("D"));

// Braces        "{7febf16f-651b-43b0-a5e3-0da8da49e90d}"
Console.WriteLine(guid.ToString("B"));

// Parentheses   "(7febf16f-651b-43b0-a5e3-0da8da49e90d)"
Console.WriteLine(guid.ToString("P"));

// Hex           "{0x7febf16f,0x651b,0x43b0{0xa5,0xe3,0x0d,0xa8,0xda,0x49,0xe9,0x0d}}"
Console.WriteLine(guid.ToString("X"));
```

Creazione di una guida

Questi sono i modi più comuni per creare un'istanza di Guid:

- Creazione di un guid vuoto (`00000000-0000-0000-0000-000000000000`):

```
Guid g = Guid.Empty;
Guid g2 = new Guid();
```

- Creazione di una nuova guida (pseudocasuale):

```
Guid g = Guid.NewGuid();
```

- Creazione di Guidi con un valore specifico:

```
Guid g = new Guid("0b214de7-8958-4956-8eed-28f9ba2c47c6");
Guid g2 = new Guid("0b214de7895849568eed28f9ba2c47c6");
Guid g3 = Guid.Parse("0b214de7-8958-4956-8eed-28f9ba2c47c6");
```

Dichiarazione di un GUID nullable

Come altri tipi di valore, anche il GUID ha un tipo nullable che può assumere valore nullo.

Dichiarazione:

```
Guid? myGuidVar = null;
```

Ciò è particolarmente utile quando si recuperano dati dal database quando esiste la possibilità che il valore di una tabella sia NULL.

Leggi guid online: https://riptutorial.com/it/csharp/topic/1153/guid

Capitolo 66: I delegati

Osservazioni

Sommario

Un **tipo delegato** è un tipo che rappresenta una particolare firma del metodo. Un'istanza di questo tipo si riferisce a un metodo particolare con una firma corrispondente. I parametri del metodo possono avere tipi delegati e quindi questo metodo deve essere passato un riferimento a un altro metodo, che può quindi essere invocato

Tipi di delegati integrati: `Action<...>` , `Predicate<T>` e `Func<...,TResult>`

Il namespace `System` contiene `Action<...>` , `Predicate<T>` e `Func<...,TResult>` delegates, dove "..." rappresenta tra 0 e 16 parametri di tipo generico (per 0 parametri, `Action` is not- generico).

`Func` rappresenta i metodi con un tipo di ritorno che corrisponde a `TResult` e `Action` rappresenta i metodi senza un valore di ritorno (void). In entrambi i casi, i parametri di tipo generico aggiuntivi corrispondono, in ordine, ai parametri del metodo.

`Predicate` rappresenta un metodo con tipo di ritorno booleano, T è un parametro di input.

Tipi di delegati personalizzati

I tipi di delegato con nome possono essere dichiarati utilizzando la parola chiave `delegate` .

Invocazione di delegati

I delegati possono essere richiamati usando la stessa sintassi dei metodi: il nome dell'istanza delegata, seguito da parentesi contenenti qualsiasi parametro.

Assegnazione ai delegati

I delegati possono essere assegnati nei seguenti modi:

- Assegnazione di un metodo denominato
- Assegnare un metodo anonimo usando un lambda
- Assegnazione di un metodo denominato utilizzando la parola chiave `delegate` .

Combinare i delegati

Più oggetti delegati possono essere assegnati a un'istanza delegata utilizzando l'operatore `+` . L'operatore `-` può essere utilizzato per rimuovere un componente delegato da un altro delegato.

Examples

Riferimenti sottostanti dei delegati del metodo denominato

Quando si assegnano metodi denominati a delegati, si riferiscono allo stesso oggetto sottostante se:

- Sono lo stesso metodo di istanza, nella stessa istanza di una classe
- Sono lo stesso metodo statico su una classe

```
public class Greeter
{
    public void WriteInstance()
    {
        Console.WriteLine("Instance");
    }

    public static void WriteStatic()
    {
        Console.WriteLine("Static");
    }
}

// ...

Greeter greeter1 = new Greeter();
Greeter greeter2 = new Greeter();

Action instance1 = greeter1.WriteInstance;
Action instance2 = greeter2.WriteInstance;
Action instance1Again = greeter1.WriteInstance;

Console.WriteLine(instance1.Equals(instance2)); // False
Console.WriteLine(instance1.Equals(instance1Again)); // True

Action @static = Greeter.WriteStatic;
Action staticAgain = Greeter.WriteStatic;

Console.WriteLine(@static.Equals(staticAgain)); // True
```

Dichiarazione di un tipo di delegato

La seguente sintassi crea un tipo `delegate` con nome `NumberInOutDelegate` , che rappresenta un metodo che accetta un `int` e restituisce un `int` .

```
public delegate int NumberInOutDelegate(int input);
```

Questo può essere usato come segue:

```
public static class Program
{
    static void Main()
    {
        NumberInOutDelegate square = MathDelegates.Square;
        int answer1 = square(4);
        Console.WriteLine(answer1); // Will output 16

        NumberInOutDelegate cube = MathDelegates.Cube;
        int answer2 = cube(4);
        Console.WriteLine(answer2); // Will output 64
    }
}

public static class MathDelegates
{
    static int Square (int x)
    {
        return x*x;
    }

    static int Cube (int x)
    {
        return x*x*x;
    }
}
```

L' `example` esempio delegato è eseguito nello stesso modo del `Square` metodo. Un'istanza delegata funge letteralmente da delegato per il chiamante: il chiamante richiama il delegato, quindi il delegato chiama il metodo di destinazione. Questo indiretto disaccoppia il chiamante dal metodo di destinazione.

È possibile dichiarare un tipo di delegato **generico** e in tal caso è possibile specificare che il tipo è covariant (`out`) o controvariante (`in`) in alcuni argomenti di tipo. Per esempio:

```
public delegate TTo Converter<in TFrom, out TTo>(TFrom input);
```

Come altri tipi generici, i tipi di delegati generici possono avere vincoli, come nel `where TFrom : struct, IConvertible where TTo : new()` .

Evitare la co- e la contravarianza per i tipi di delegati che devono essere utilizzati per i delegati multicast, come i tipi di gestori di eventi. Questo perché la concatenazione (`+`) può fallire se il tipo di runtime è diverso dal tipo in fase di compilazione a causa della varianza. Ad esempio, evitare:

```
public delegate void EventHandler<in TEventArgs>(object sender, TEventArgs e);
```

Invece, usa un tipo generico invariante:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Sono supportati anche i delegati in cui alcuni parametri sono modificati da `ref` o `out` , come in:

```
public delegate bool TryParser<T>(string input, out T result);
```

(Esempio: `TryParser<decimal> example = decimal.TryParse;`), o delegati in cui l'ultimo parametro ha il modificatore `params` . I tipi di delegati possono avere parametri opzionali (fornire valori predefiniti). I tipi di delegati possono usare tipi di puntatore come `int*` o `char*` nelle loro firme o tipi di ritorno (usa una parola chiave `unsafe`). Un tipo di delegato e i relativi parametri possono contenere attributi personalizzati.

Il Func , Azione e Predicato tipi di delegati

Il namespace System contiene `Func<..., TResult>` tipi di delegati con tra 0 e 15 parametri generici, restituendo il tipo `TResult` .

```
private void UseFunc(Func<string> func)
{
    string output = func(); // Func with a single generic type parameter returns that type
    Console.WriteLine(output);
}

private void UseFunc(Func<int, int, string> func)
{
    string output = func(4, 2); // Func with multiple generic type parameters takes all but
the first as parameters of that type
    Console.WriteLine(output);
}
```

Il namespace System contiene anche i tipi di `Action<...>` delegate con un numero diverso di parametri generici (da 0 a 16). È simile a `Func<T1, .., Tn>` , ma restituisce sempre `void` .

```
private void UseAction(Action action)
{
    action(); // The non-generic Action has no parameters
}

private void UseAction(Action<int, string> action)
{
    action(4, "two"); // The generic action is invoked with parameters matching its type
arguments
}
```

`Predicate<T>` è anche una forma di `Func` ma restituirà sempre `bool` . Un predicato è un modo per specificare un criterio personalizzato. A seconda del valore dell'input e della logica definita all'interno del predicato, restituirà `true` o `false` . `Predicate<T>` si comporta quindi allo stesso modo di `Func<T, bool>` ed entrambi possono essere inizializzati e utilizzati allo stesso modo.

```
Predicate<string> predicate = s => s.StartsWith("a");
Func<string, bool> func = s => s.StartsWith("a");

// Both of these return true
var predicateReturnsTrue = predicate("abc");
```

```
var funcReturnsTrue = func("abc");

// Both of these return false
var predicateReturnsFalse = predicate("xyz");
var funcReturnsFalse = func("xyz");
```

La scelta se utilizzare `Predicate<T>` o `Func<T, bool>` è davvero una questione di opinione. `Predicate<T>` è probabilmente più espressivo delle intenzioni dell'autore, mentre `Func<T, bool>` è probabilmente familiare a una percentuale maggiore di sviluppatori C #.

In aggiunta a ciò, ci sono alcuni casi in cui è disponibile solo una delle opzioni, specialmente quando si interagisce con un'altra API. Ad esempio, `List<T>` e `Array<T>` genere accettano `Predicate<T>` per i loro metodi, mentre la maggior parte delle estensioni LINQ accettano solo `Func<T, bool>` .

Assegnazione di un metodo denominato a un delegato

I metodi con nome possono essere assegnati ai delegati con firme corrispondenti:

```
public static class Example
{
    public static int AddOne(int input)
    {
        return input + 1;
    }
}


Func<int,int> addOne = Example.AddOne
```

`Example.AddOne` prende un `int` e restituisce un `int` , la sua firma corrisponde al delegato `Func<int,int>` . `Example.AddOne` può essere assegnato direttamente ad `addOne` perché hanno firme corrispondenti.

Delegare l'uguaglianza

Chiamare `.Equals()` su un delegato viene confrontato per uguaglianza di riferimento:

```
Action action1 = () => Console.WriteLine("Hello delegates");
Action action2 = () => Console.WriteLine("Hello delegates");
Action action1Again = action1;

Console.WriteLine(action1.Equals(action1)) // True
Console.WriteLine(action1.Equals(action2)) // False
Console.WriteLine(action1Again.Equals(action1)) // True
```

Queste regole si applicano anche quando si fa `+=` o `-=` su un delegato multicast, ad esempio quando si sottoscrive e si annulla la sottoscrizione dagli eventi.

Assegnazione a un delegato di lambda

Lambdas può essere utilizzato per creare metodi anonimi da assegnare a un delegato:

```
Func<int,int> addOne = x => x+1;
```

Si noti che la dichiarazione esplicita di tipo è richiesta quando si crea una variabile in questo modo:

```
var addOne = x => x+1; // Does not work
```

Passando delegati come parametri

I delegati possono essere utilizzati come puntatori di funzione tipizzati:

```
class FuncAsParameters
{
  public void Run()
  {
    DoSomething(ErrorHandler1);
    DoSomething(ErrorHandler2);
  }

  public bool ErrorHandler1(string message)
  {
    Console.WriteLine(message);
    var shouldWeContinue = ...
    return shouldWeContinue;
  }

  public bool ErrorHandler2(string message)
  {
    // ...Write message to file...
    var shouldWeContinue = ...
    return shouldWeContinue;
  }

  public void DoSomething(Func<string, bool> errorHandler)
  {
    // In here, we don't care what handler we got passed!
    ...
    if (...error...)
    {
      if (!errorHandler("Some error occurred!"))
      {
        // The handler decided we can't continue
        return;
      }
    }
  }
}
```

Combina delegati (delegati multicast)

Addizione `+` e sottrazione `-` operazioni possono essere utilizzate per combinare le istanze delegate. Il delegato contiene un elenco dei delegati assegnati.

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace DelegatesExample {
    class MainClass {
        private delegate void MyDelegate(int a);

        private static void PrintInt(int a) {
            Console.WriteLine(a);
        }

        private static void PrintType<T>(T a) {
            Console.WriteLine(a.GetType());
        }

        public static void Main (string[] args)
        {
            MyDelegate d1 = PrintInt;
            MyDelegate d2 = PrintType;

            // Output:
            // 1
            d1(1);

            // Output:
            // System.Int32
            d2(1);

            MyDelegate d3 = d1 + d2;
            // Output:
            // 1
            // System.Int32
            d3(1);

            MyDelegate d4 = d3 - d2;
            // Output:
            // 1
            d4(1);

            // Output:
            // True
            Console.WriteLine(d1 == d4);
        }
    }
}
```

In questo esempio `d3` è una combinazione di delegati `d1` e `d2` , quindi quando viene chiamato il programma emette sia `1` che le stringhe `System.Int32` .

Combinazione di delegati con tipi di **reso non void** :

Se un delegato multicast ha un tipo di `nonvoid` , il chiamante riceve il valore di ritorno dall'ultimo metodo da richiamare. I metodi precedenti sono ancora chiamati, ma i loro valori di ritorno sono scartati.

```
    class Program
```

```
    {
        public delegate int Transformer(int x);

        static void Main(string[] args)
        {
            Transformer t = Square;
            t += Cube;
            Console.WriteLine(t(2));  // O/P 8
        }

        static int Square(int x) { return x * x; }

        static int Cube(int x) { return x*x*x; }
    }
```

`t(2)` chiamerà prima `Square` e poi `Cube` . Il valore di ritorno di Square viene scartato e restituisce il valore dell'ultimo metodo, ovvero `Cube` viene mantenuto.

Sicuro invocare delegato multicast

Hai mai desiderato chiamare un delegato multicast ma vuoi che venga chiamato l'intero elenco di invocazioni anche se si verifica un'eccezione in una catena qualsiasi. Quindi sei fortunato, ho creato un metodo di estensione che fa proprio questo, lanciando un `AggregateException` solo dopo che l'esecuzione dell'intero elenco è stata completata:

```
public static class DelegateExtensions
{
    public static void SafeInvoke(this Delegate del,params object[] args)
    {
        var exceptions = new List<Exception>();

        foreach (var handler in del.GetInvocationList())
        {
            try
            {
                handler.Method.Invoke(handler.Target, args);
            }
            catch (Exception ex)
            {
                exceptions.Add(ex);
            }
        }

        if(exceptions.Any())
        {
            throw new AggregateException(exceptions);
        }
    }
}

public class Test
{
    public delegate void SampleDelegate();

    public void Run()
    {
        SampleDelegate delegateInstance = this.Target2;
```

```
        delegateInstance += this.Target1;

        try
        {
            delegateInstance.SafeInvoke();
        }
        catch(AggregateException ex)
        {
            // Do any exception handling here
        }
    }

    private void Target1()
    {
        Console.WriteLine("Target 1 executed");
    }

    private void Target2()
    {
        Console.WriteLine("Target 2 executed");
        throw new Exception();
    }
}
```

Questo produce:

```
Target 2 executed
Target 1 executed
```

`SaveInvoke` direttamente, senza `SaveInvoke` , eseguirà solo Target 2.

Chiusura in un delegato

Le chiusure sono metodi anonimi incorporati che hanno la possibilità di utilizzare le variabili del metodo `Parent` e altri metodi anonimi che sono definiti nell'ambito del genitore.

> In sostanza, una chiusura è un blocco di codice che può essere eseguito in un secondo momento, ma che mantiene l'ambiente in cui è stato creato, cioè può ancora utilizzare le variabili locali ecc del metodo che lo ha creato, anche dopo il metodo ha terminato l'esecuzione. **- Jon Skeet**

```
delegate int testDel();
static void Main(string[] args)
{
    int foo = 4;
    testDel myClosure = delegate()
    {
        return foo;
    };
    int bar = myClosure();

}
```

Esempio tratto da Closures in .NET .

Incapsulando trasformazioni in funzioni

```
public class MyObject{
    public DateTime? TestDate { get; set; }

    public Func<MyObject, bool> DateIsValid = myObject => myObject.TestDate.HasValue &&
myObject.TestDate > DateTime.Now;

    public void DoSomething(){
        //We can do this:
        if(this.TestDate.HasValue && this.TestDate > DateTime.Now){
            CallAnotherMethod();
        }

        //or this:
        if(DateIsValid(this)){
            CallAnotherMethod();
        }
    }
}
```

Nello spirito di una codifica pulita, l'incapsulamento di verifiche e trasformazioni come quella di cui sopra come Func può facilitare la lettura e la comprensione del codice. Mentre l'esempio sopra è molto semplice, e se ci fossero più proprietà DateTime ognuna con le proprie regole di convalida diverse e volessimo controllare diverse combinazioni? Semplici funzioni di una riga che ciascuna ha stabilito la logica di ritorno possono essere entrambe leggibili e ridurre l'apparente complessità del codice. Considera le chiamate di Func qui sotto e immagina quanto più codice ingombrerebbe il metodo:

```
public void CheckForIntegrity(){
    if(ShipDateIsValid(this) && TestResultsHaveBeenIssued(this) && !TestResultsFail(this)){
        SendPassingTestNotification();
    }
}
```

Leggi I delegati online: https://riptutorial.com/it/csharp/topic/1194/i-delegati

Capitolo 67: ICloneable

Sintassi

- oggetto ICloneable.Clone () {return Clone (); } // Implementazione privata del metodo di interfaccia che utilizza la nostra funzione pubblica personalizzata Clone ().
- pubblico Foo Clone () {return new Foo (this); } // Il metodo clone pubblico dovrebbe utilizzare la logica del costruttore di copie.

Osservazioni

Il `CLR` richiede un `object Clone()` definizione del metodo `object Clone()` che non sia sicuro da testo. È prassi comune sovrascrivere questo comportamento e definire un metodo sicuro per tipo che restituisce una copia della classe contenente.

Spetta all'autore decidere se la clonazione significa solo copia superficiale o copia profonda. Per strutture immutabili contenenti riferimenti si consiglia di fare una copia profonda. Poiché le classi sono riferimenti a se stessi, probabilmente è meglio implementare una copia superficiale.

NOTA: In `C#` un metodo di interfaccia può essere implementato privatamente con la sintassi mostrata sopra.

Examples

Implementazione ICloneable in una classe

Implementare `ICloneable` in una classe con una svolta. Esporre un `Clone()` tipo pubblico sicuro e implementare l' `object Clone()` privatamente.

```
public class Person : ICloneable
{
    // Contents of class
    public string Name { get; set; }
    public int Age { get; set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        this.Name=other.Name;
        this.Age=other.Age;
    }

    #region ICloneable Members
    // Type safe Clone
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
```

```
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

Successivamente sarà usato come segue:

```
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);

    bob.Age=56;
    Debug.Assert(bob.Age!=bob.Age);
}
```

Si noti che la modifica dell'età di `bob` non modifica l'età di `bob_clone` . Questo perché il design utilizza la clonazione anziché l'assegnazione di variabili (di riferimento).

Implementazione ICloneable in una struttura

L'implementazione di ICloneable per una struttura non è in genere necessaria poiché le strutture eseguono una copia membro con l'operatore di assegnazione `=` . Ma il progetto potrebbe richiedere l'implementazione di un'altra interfaccia che eredita da `ICloneable` .

Un altro motivo potrebbe essere se la struttura contiene un tipo di riferimento (o un array) che avrebbe bisogno anche di una copia.

```
// Structs are recommended to be immutable objects
[ImmutableObject(true)]
public struct Person : ICloneable
{
    // Contents of class
    public string Name { get; private set; }
    public int Age { get; private set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        // The assignment operator copies all members
        this=other;
    }

    #region ICloneable Members
    // Type safe Clone
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
    object ICloneable.Clone()
    {
```

```
        return Clone();
    }
    #endregion
}
```

Successivamente sarà usato come segue:

```
static void Main(string[] args)
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);
}
```

Leggi ICloneable online: https://riptutorial.com/it/csharp/topic/7917/icloneable

Capitolo 68: IComparable

Examples

Ordina versioni

Classe:

```
public class Version : IComparable<Version>
{
    public int[] Parts { get; }

    public Version(string value)
    {
        if (value == null)
            throw new ArgumentNullException();
        if (!Regex.IsMatch(value, @"^[0-9]+(\.[0-9]+)*$"))
            throw new ArgumentException("Invalid format");
        var parts = value.Split('.');
        Parts = new int[parts.Length];
        for (var i = 0; i < parts.Length; i++)
            Parts[i] = int.Parse(parts[i]);
    }

    public override string ToString()
    {
        return string.Join(".", Parts);
    }

    public int CompareTo(Version that)
    {
        if (that == null) return 1;
        var thisLength = this.Parts.Length;
        var thatLength = that.Parts.Length;
        var maxLength = Math.Max(thisLength, thatLength);
        for (var i = 0; i < maxLength; i++)
        {
            var thisPart = i < thisLength ? this.Parts[i] : 0;
            var thatPart = i < thatLength ? that.Parts[i] : 0;
            if (thisPart < thatPart) return -1;
            if (thisPart > thatPart) return 1;
        }
        return 0;
    }
}
```

Test:

```
Version a, b;

a = new Version("4.2.1");
b = new Version("4.2.6");
a.CompareTo(b); // a < b : -1

a = new Version("2.8.4");
```

```
b = new Version("2.8.0");
a.CompareTo(b); // a > b : 1

a = new Version("5.2");
b = null;
a.CompareTo(b); // a > b : 1

a = new Version("3");
b = new Version("3.6");
a.CompareTo(b); // a < b : -1

var versions = new List<Version>
{
    new Version("2.0"),
    new Version("1.1.5"),
    new Version("3.0.10"),
    new Version("1"),
    null,
    new Version("1.0.1")
};

versions.Sort();

foreach (var version in versions)
    Console.WriteLine(version?.ToString() ?? "NULL");
```

Produzione:

NULLO
1
1.0.1
1.1.5
2.0
3.0.10

demo:

Demo live su Ideone

Leggi IComparable online: https://riptutorial.com/it/csharp/topic/4222/icomparable

Capitolo 69: Identità ASP.NET

introduzione

Esercitazioni su Asp.net Identity come gestione degli utenti, gestione dei ruoli, creazione di token e altro.

Examples

Come implementare il token di reimpostazione della password nell'identità di asp.net utilizzando user manager.

1. Crea una nuova cartella chiamata MyClasses e crea e aggiungi la seguente classe

```
public class GmailEmailService:SmtpClient
{
    // Gmail user-name
    public string UserName { get; set; }

    public GmailEmailService() :
        base(ConfigurationManager.AppSettings["GmailHost"],
Int32.Parse(ConfigurationManager.AppSettings["GmailPort"]))
    {
        //Get values from web.config file:
        this.UserName = ConfigurationManager.AppSettings["GmailUserName"];
        this.EnableSsl = Boolean.Parse(ConfigurationManager.AppSettings["GmailSsl"]);
        this.UseDefaultCredentials = false;
        this.Credentials = new System.Net.NetworkCredential(this.UserName,
ConfigurationManager.AppSettings["GmailPassword"]);
    }
}
```

2. Configura la tua classe di identità

```
public async Task SendAsync(IdentityMessage message)
{
    MailMessage email = new MailMessage(new MailAddress("youremailadress@domain.com",
"(any subject here)"),
    new MailAddress(message.Destination));
    email.Subject = message.Subject;
    email.Body = message.Body;

    email.IsBodyHtml = true;

    GmailEmailService mailClient = new GmailEmailService();
    await mailClient.SendMailAsync(email);
}
```

3. Aggiungi le tue credenziali a web.config. Non ho usato Gmail in questa parte perché l'utilizzo di Gmail è bloccato sul mio posto di lavoro e funziona perfettamente.

```
<add key="GmailUserName" value="youremail@yourdomain.com"/>
<add key="GmailPassword" value="yourPassword"/>
<add key="GmailHost" value="yourServer"/>
<add key="GmailPort" value="yourPort"/>
<add key="GmailSsl" value="chooseTrueOrFalse"/>
<!--Smptp Server (confirmations emails)-->
```

4. Apporta le modifiche necessarie al tuo Account Controller. Aggiungi il seguente codice evidenziato.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.DataProtection;
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using MYPROJECT.Models;

namespace MYPROJECT.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private ApplicationSignInManager _signInManager;
        private ApplicationUserManager _userManager;
        ApplicationDbContext _context;
        DpapiDataProtectionProvider provider = new DpapiDataProtectionProvider("MYPROJECT");
        EmailService EmailService = new EmailService();


        public AccountController()
            : this(new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new ApplicationDbContext
        {
            _context = new ApplicationDbContext();

        }

        public AccountController(UserManager<ApplicationUser> userManager, ApplicationSignInManager signIn
        {
            UserManager = userManager;
            SignInManager = signInManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                    provider.Create("EmailConfirmation"));

        }

        private AccountController(UserManager<ApplicationUser> userManager)
        {
            UserManager = userManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                    provider.Create("EmailConfirmation"));
        }

        public UserManager<ApplicationUser> UserManager { get; private set; }

        public ApplicationSignInManager SignInManager
        {
            get
            {
                return _signInManager ?? HttpContext.GetOwinContext().Get<ApplicationSignInManager>();
            }
            private set
            {
                _signInManager = value;
            }
        }
```

```
//
// GET: /Account/ForgotPassword
[AllowAnonymous]
public ActionResult ForgotPassword()
{
    return View();
}

//
// POST: /Account/ForgotPassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {

        var user = await UserManager.FindByNameAsync(model.UserName);

        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))
        {
            // Don't reveal that the user does not exist or is not confirmed
            return View("ForgotPasswordConfirmation");
        }

        // For more information on how to enable account confirmation and password reset please visit http
        // Send an email with this link
        string code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);
        code = HttpUtility.UrlEncode(code);
        var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = HttpUtility
        await EmailService.SendAsync(new IdentityMessage
        {
            Body = "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>",
            Destination = user.Email,
            Subject = "Reset Password"
        });
        //await UserManager.SendEmailAsync(user.Id, "Reset Password", "Please reset your password by clicki
        return RedirectToAction("ForgotPasswordConfirmation", "Account");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Compilare quindi eseguire. Saluti!

Leggi Identità ASP.NET online: https://riptutorial.com/it/csharp/topic/9577/identita-asp-net

Capitolo 70: IEnumerable

introduzione

`IEnumerable` è l'interfaccia di base per tutte le raccolte non generiche come ArrayList che possono essere enumerate. `IEnumerator<T>` è l'interfaccia di base per tutti gli enumeratori generici come Elenco <>.

`IEnumerable` è un'interfaccia che implementa il metodo `GetEnumerator` . Il metodo `GetEnumerator` restituisce un `IEnumerator` che fornisce opzioni per scorrere la raccolta come foreach.

Osservazioni

IEnumerable è l'interfaccia di base per tutte le raccolte non generiche che possono essere enumerate

Examples

IEnumerable

Nella sua forma più elementare, un oggetto che implementa IEnumerable rappresenta una serie di oggetti. Gli oggetti in questione possono essere iterati utilizzando la parola chiave c `foreach` .

Nell'esempio seguente, la `sequenceOfNumbers` oggettiOfNumbers implementa IEnumerable. Rappresenta una serie di numeri interi. Il ciclo `foreach` esegue iterate a turno ciascuna.

```
int AddNumbers(IEnumerable<int> sequenceOfNumbers) {
    int returnValue = 0;
    foreach(int i in sequenceOfNumbers) {
        returnValue += i;
    }
    return returnValue;
}
```

IEnumerable with Enumerator personalizzato

L'implementazione dell'interfaccia IEnumerable consente di classificare le classi allo stesso modo delle raccolte BCL. Ciò richiede l'estensione della classe Enumerator che tiene traccia dello stato dell'enumerazione.

A parte l'iterazione su una raccolta standard, gli esempi includono:

- Utilizzo di intervalli di numeri basati su una funzione anziché su una raccolta di oggetti
- Implementazione di algoritmi di iterazione diversi su raccolte, come DFS o BFS su una raccolta di grafici

```
public static void Main(string[] args) {

    foreach (var coffee in new CoffeeCollection()) {
        Console.WriteLine(coffee);
    }
}

public class CoffeeCollection : IEnumerable {
    private CoffeeEnumerator enumerator;

    public CoffeeCollection() {
        enumerator = new CoffeeEnumerator();
    }

    public IEnumerator GetEnumerator() {
        return enumerator;
    }

    public class CoffeeEnumerator : IEnumerator {
        string[] beverages = new string[3] { "espresso", "macchiato", "latte" };
        int currentIndex = -1;

        public object Current {
            get {
                return beverages[currentIndex];
            }
        }

        public bool MoveNext() {
            currentIndex++;

            if (currentIndex < beverages.Length) {
                return true;
            }

            return false;
        }

        public void Reset() {
            currentIndex = 0;
        }
    }
}
```

Leggi IEnumerable online: https://riptutorial.com/it/csharp/topic/2220/ienumerable

Capitolo 71: ILGenerator

Examples

Crea un DynamicAssembly che contiene un metodo di supporto UnixTimestamp

Questo esempio mostra l'utilizzo di ILGenerator generando codice che fa uso di membri già esistenti e nuovi creati oltre alla gestione di base delle eccezioni. Il codice seguente emette un DynamicAssembly che contiene un equivalente a questo codice c #:

```
public static class UnixTimeHelper
{
    private readonly static DateTime EpochTime = new DateTime(1970, 1, 1);

    public static int UnixTimestamp(DateTime input)
    {
        int totalSeconds;
        try
        {
            totalSeconds =
checked((int)input.Subtract(UnixTimeHelper.EpochTime).TotalSeconds);
        }
        catch (OverflowException overflowException)
        {
            throw new InvalidOperationException("It's to late for an Int32 timestamp.",
overflowException);
        }
        return totalSeconds;
    }
}
```

```
//Get the required methods
var dateTimeCtor = typeof (DateTime)
    .GetConstructor(new[] {typeof (int), typeof (int), typeof (int)});
var dateTimeSubstract = typeof (DateTime)
    .GetMethod(nameof(DateTime.Subtract), new[] {typeof (DateTime)});
var timeSpanSecondsGetter = typeof (TimeSpan)
    .GetProperty(nameof(TimeSpan.TotalSeconds)).GetGetMethod();
var invalidOperationCtor = typeof (InvalidOperationException)
    .GetConstructor(new[] {typeof (string), typeof (Exception)});

if (dateTimeCtor == null || dateTimeSubstract == null ||
    timeSpanSecondsGetter == null || invalidOperationCtor == null)
{
    throw new Exception("Could not find a required Method, can not create Assembly.");
}

//Setup the required members
var an = new AssemblyName("UnixTimeAsm");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(an,
AssemblyBuilderAccess.RunAndSave);
var dynMod = dynAsm.DefineDynamicModule(an.Name, an.Name + ".dll");
```

```
var dynType = dynMod.DefineType("UnixTimeHelper",
    TypeAttributes.Abstract | TypeAttributes.Sealed | TypeAttributes.Public);

var epochTimeField = dynType.DefineField("EpochStartTime", typeof (DateTime),
    FieldAttributes.Private | FieldAttributes.Static | FieldAttributes.InitOnly);

var cctor =
    dynType.DefineConstructor(
        MethodAttributes.Private | MethodAttributes.HideBySig | MethodAttributes.SpecialName |
        MethodAttributes.RTSpecialName | MethodAttributes.Static, CallingConventions.Standard,
        Type.EmptyTypes);

var cctorGen = cctor.GetILGenerator();
cctorGen.Emit(OpCodes.Ldc_I4, 1970); //Load the DateTime constructor arguments onto the stack
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Newobj, dateTimeCtor); //Call the constructor
cctorGen.Emit(OpCodes.Stsfld, epochTimeField); //Store the object in the static field
cctorGen.Emit(OpCodes.Ret);

var unixTimestampMethod = dynType.DefineMethod("UnixTimestamp",
    MethodAttributes.Public | MethodAttributes.HideBySig | MethodAttributes.Static,
    CallingConventions.Standard, typeof (int), new[] {typeof (DateTime)});

unixTimestampMethod.DefineParameter(1, ParameterAttributes.None, "input");

var methodGen = unixTimestampMethod.GetILGenerator();
methodGen.DeclareLocal(typeof (TimeSpan));
methodGen.DeclareLocal(typeof (int));
methodGen.DeclareLocal(typeof (OverflowException));

methodGen.BeginExceptionBlock(); //Begin the try block
methodGen.Emit(OpCodes.Ldarga_S, (byte) 0); //To call a method on a struct we need to load the
address of it
methodGen.Emit(OpCodes.Ldsfld, epochTimeField);
    //Load the object of the static field we created as argument for the following call
methodGen.Emit(OpCodes.Call, dateTimeSubstract); //Call the substract method on the input
DateTime
methodGen.Emit(OpCodes.Stloc_0); //Store the resulting TimeSpan in a local
methodGen.Emit(OpCodes.Ldloca_S, (byte) 0); //Load the locals address to call a method on it
methodGen.Emit(OpCodes.Call, timeSpanSecondsGetter); //Call the TotalSeconds Get method on the
TimeSpan
methodGen.Emit(OpCodes.Conv_Ovf_I4); //Convert the result to Int32; throws an exception on
overflow
methodGen.Emit(OpCodes.Stloc_1); //store the result for returning later
//The leave instruction to jump behind the catch block will be automatically emitted
methodGen.BeginCatchBlock(typeof (OverflowException)); //Begin the catch block
//When we are here, an OverflowException was thrown, that is now on the stack
methodGen.Emit(OpCodes.Stloc_2); //Store the exception in a local.
methodGen.Emit(OpCodes.Ldstr, "It's to late for an Int32 timestamp.");
    //Load our error message onto the stack
methodGen.Emit(OpCodes.Ldloc_2); //Load the exception again
methodGen.Emit(OpCodes.Newobj, invalidOperationCtor);
    //Create an InvalidOperationException with our message and inner Exception
methodGen.Emit(OpCodes.Throw); //Throw the created exception
methodGen.EndExceptionBlock(); //End the catch block
//When we are here, everything is fine
methodGen.Emit(OpCodes.Ldloc_1); //Load the result value
methodGen.Emit(OpCodes.Ret); //Return it
```

```
dynType.CreateType();

dynAsm.Save(an.Name + ".dll");
```

Crea sovrascrittura del metodo

Questo esempio mostra come sovrascrivere il metodo `ToString` nella classe generata

```
// create an Assembly and new type
var name = new AssemblyName("MethodOverriding");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(name,
AssemblyBuilderAccess.RunAndSave);
var dynModule = dynAsm.DefineDynamicModule(name.Name, $"{name.Name}.dll");
var typeBuilder = dynModule.DefineType("MyClass", TypeAttributes.Public |
TypeAttributes.Class);

// define a new method
var toStr = typeBuilder.DefineMethod(
    "ToString", // name
    MethodAttributes.Public | MethodAttributes.Virtual, // modifiers
    typeof(string), // return type
    Type.EmptyTypes); // argument types
var ilGen = toStr.GetILGenerator();
ilGen.Emit(OpCodes.Ldstr, "Hello, world!");
ilGen.Emit(OpCodes.Ret);

// set this method as override of object.ToString
typeBuilder.DefineMethodOverride(toStr, typeof(object).GetMethod("ToString"));
var type = typeBuilder.CreateType();

// now test it:
var instance = Activator.CreateInstance(type);
Console.WriteLine(instance.ToString());
```

Leggi ILGenerator online: https://riptutorial.com/it/csharp/topic/667/ilgenerator

Capitolo 72: Immutabilità

Examples

Classe System.String

In C # (e .NET) una stringa è rappresentata dalla classe System.String. La parola chiave `string` è un alias per questa classe.

La classe System.String è immutabile, ovvero una volta creato il suo stato non può essere modificato.

Quindi tutte le operazioni eseguite su una stringa come Substring, Remove, Replace, concatenation usando `+` operatore etc creeranno una nuova stringa e la restituiranno.

Vedi il seguente programma per dimostrazione -

```
string str = "mystring";
string newString = str.Substring(3);
Console.WriteLine(newString);
Console.WriteLine(str);
```

Questo stamperà rispettivamente `string` e `mystring` .

Archi e immutabilità

I tipi immutabili sono tipi che, una volta modificati, creano una nuova versione dell'oggetto in memoria, anziché modificare l'oggetto esistente nella memoria. L'esempio più semplice di questo è il tipo di `string` incorporato.

Prendendo il seguente codice, che aggiunge "mondo" alla parola "Ciao"

```
string myString = "hello";
myString += " world";
```

Ciò che sta accadendo in memoria in questo caso è che un nuovo oggetto viene creato quando si aggiunge alla `string` nella seconda riga. Se si esegue questa operazione come parte di un loop di grandi dimensioni, è possibile che ciò causi problemi di prestazioni nell'applicazione.

L'equivalente mutabile per una `string` è un `StringBuilder`

Prendendo il seguente codice

```
StringBuilder myStringBuilder = new StringBuilder("hello");
myStringBuilder.append(" world");
```

Quando lo esegui, stai modificando l'oggetto `StringBuilder` stesso nella memoria.

Leggi Immutabilità online: https://riptutorial.com/it/csharp/topic/1863/immutabilita

Capitolo 73: Implementazione del modello di design Flyweight

Examples

Implementazione della mappa nel gioco RPG

Il peso piuma è uno dei modelli di progettazione strutturale. Viene utilizzato per ridurre la quantità di memoria utilizzata condividendo il maggior numero possibile di dati con oggetti simili. Questo documento ti insegnerà come usare correttamente Flyweight DP.

Lascia che ti spieghi l'idea su un semplice esempio. Immagina di lavorare su un gioco di ruolo e devi caricare un file enorme contenente alcuni personaggi. Per esempio:

- `#` è erba Puoi camminarci sopra.
- `$` è il punto di partenza
- `@` è rock. Non puoi camminarci sopra.
- `%` è scrigno del tesoro

Esempio di una mappa:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@##############@@@@@######@#$@@@

@##############@@######@####@@@

@########%######@############@@@

@############################@

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Poiché questi oggetti hanno caratteristiche simili, non è necessario creare oggetti separati per ogni campo mappa. Ti mostrerò come usare il peso mosca.

Definiamo un'interfaccia che i nostri campi implementeranno:

```
public interface IField
{
    string Name { get; }
    char Mark { get; }
    bool CanWalk { get; }
    FieldType Type { get; }
}
```

Ora possiamo creare classi che rappresentano i nostri campi. Dobbiamo anche identificarli in qualche modo (ho usato un'enumerazione):

```
public enum FieldType
{
    GRASS,
    ROCK,
    START,
    CHEST
}
public class Grass : IField
{
    public string Name { get { return "Grass"; } }
    public char Mark { get { return '#'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.GRASS; } }
}
public class StartingPoint : IField
{
    public string Name { get { return "Starting Point"; } }
    public char Mark { get { return '$'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.START; } }
}
public class Rock : IField
{
    public string Name { get { return "Rock"; } }
    public char Mark { get { return '@'; } }
    public bool CanWalk { get { return false; } }
    public FieldType Type { get { return FieldType.ROCK; } }
}
public class TreasureChest : IField
{
    public string Name { get { return "Treasure Chest"; } }
    public char Mark { get { return '%'; } }
    public bool CanWalk { get { return true; } } // you can approach it
    public FieldType Type { get { return FieldType.CHEST; } }
}
```

Come ho detto, non è necessario creare un'istanza separata per ogni campo. Dobbiamo creare un **repository** di campi. L'essenza di Flyweight DP è che creiamo dinamicamente un oggetto solo se ne abbiamo bisogno e non esiste ancora nel nostro repository, o lo restituiamo se già esiste. Scriviamo una semplice lezione che gestirà questo per noi:

```
public class FieldRepository
{
    private List<IField> lstFields = new List<IField>();

    private IField AddField(FieldType type)
    {
        IField f;
        switch(type)
        {
            case FieldType.GRASS: f = new Grass(); break;
            case FieldType.ROCK: f = new Rock(); break;
            case FieldType.START: f = new StartingPoint(); break;
            case FieldType.CHEST:
            default: f = new TreasureChest(); break;
        }
        lstFields.Add(f); //add it to repository
        Console.WriteLine("Created new instance of {0}", f.Name);
        return f;
```

```
    }
    public IField GetField(FieldType type)
    {
        IField f = lstFields.Find(x => x.Type == type);
        if (f != null) return f;
        else return AddField(type);
    }
}
```

Grande! Ora possiamo testare il nostro codice:

```
public class Program
{
    public static void Main(string[] args)
    {
        FieldRepository f = new FieldRepository();
        IField grass = f.GetField(FieldType.GRASS);
        grass = f.GetField(FieldType.ROCK);
        grass = f.GetField(FieldType.GRASS);
    }
}
```

Il risultato nella console dovrebbe essere:

> Creata una nuova istanza di Grass
>
> Creata una nuova istanza di Rock

Ma perché l'erba appare solo una volta se volevamo ottenerla due volte? Questo perché la prima volta che chiamiamo l'istanza di erba di `GetField` non esiste nel nostro **repository** , quindi è stato creato, ma la prossima volta che abbiamo bisogno di erba esiste già, quindi lo restituiamo solo.

Leggi Implementazione del modello di design Flyweight online:
https://riptutorial.com/it/csharp/topic/4619/implementazione-del-modello-di-design-flyweight

Capitolo 74: Implementazione del pattern di progettazione di Decorator

Osservazioni

A favore dell'utilizzo di Decorator:

- è possibile aggiungere nuove funzionalità in fase di esecuzione in diverse configurazioni
- buona alternativa per l'ereditarietà
- il cliente può scegliere la configurazione che desidera utilizzare

Examples

Simulazione della caffetteria

Decoratore è uno dei modelli di progettazione strutturale. È usato per aggiungere, rimuovere o modificare il comportamento dell'oggetto. Questo documento ti insegnerà come usare Decorator DP correttamente.

Lascia che ti spieghi l'idea su un semplice esempio. Immagina di essere a Starbobs, famosa compagnia di caffè. È possibile effettuare un ordine per qualsiasi caffè desiderato - con panna e zucchero, con panna e farcitura e molte altre combinazioni! Ma la base di tutte le bevande è il caffè - bevanda scura e amara, che puoi modificare. Scriviamo un semplice programma che simula la macchina del caffè.

Innanzitutto, dobbiamo creare e astrarre una classe che descriva la nostra bevanda di base:

```
public abstract class AbstractCoffee
{
    protected AbstractCoffee k = null;

    public AbstractCoffee(AbstractCoffee k)
    {
        this.k = k;
    }

    public abstract string ShowCoffee();
}
```

Ora, creiamo degli extra, come lo zucchero, il latte e il topping. Le classi create devono implementare `AbstractCoffee` - lo decoreranno:

```
public class Milk : AbstractCoffee
{
    public Milk(AbstractCoffee c) : base(c) { }
    public override string ShowCoffee()
    {
```

```
        if (k != null)
            return k.ShowCoffee() + " with Milk";
        else return "Milk";
    }
}
public class Sugar : AbstractCoffee
{
    public Sugar(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Sugar";
        else return "Sugar";
    }
}
public class Topping : AbstractCoffee
{
    public Topping(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Topping";
        else return "Topping";
    }
}
```

Ora possiamo creare il nostro caffè preferito:

```
public class Program
{
    public static void Main(string[] args)
    {
        AbstractCoffee coffee = null; //we cant create instance of abstract class
        coffee = new Topping(coffee); //passing null
        coffee = new Sugar(coffee); //passing topping instance
        coffee = new Milk(coffee);  //passing sugar
        Console.WriteLine("Coffee with " + coffee.ShowCoffee());

    }
}
```

L'esecuzione del codice produrrà il seguente risultato:

> Caffè con Topping con zucchero e latte

Leggi Implementazione del pattern di progettazione di Decorator online: https://riptutorial.com/it/csharp/topic/4798/implementazione-del-pattern-di-progettazione-di-decorator

Capitolo 75: Implementazione di Singleton

Examples

Singleton inizializzato staticamente

```
public class Singleton
{
    private readonly static Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton Instance => instance;
}
```

Questa implementazione è thread-safe perché in questo caso l'oggetto `instance` viene inizializzato nel costruttore statico. Il CLR garantisce già che tutti i costruttori statici siano eseguiti thread-safe.

L' `instance` mutante non è un'operazione thread-safe, pertanto l'attributo `readonly` garantisce l'immutabilità dopo l'inizializzazione.

Singleton pigro e sicuro per i thread (utilizzando Double Checked Locking)

Questa versione thread-safe di un singleton era necessaria nelle prime versioni di .NET in cui l'inizializzazione `static` non era garantita per essere thread-safe. Nelle versioni più moderne del framework di solito viene preferito un singleton inizializzato staticamente perché è molto facile commettere errori di implementazione nel seguente schema.

```
public sealed class ThreadSafeSingleton
{
   private static volatile ThreadSafeSingleton instance;
   private static object lockObject = new Object();

   private ThreadSafeSingleton()
   {
   }

   public static ThreadSafeSingleton Instance
   {
      get
      {
         if (instance == null)
         {
            lock (lockObject)
            {
               if (instance == null)
               {
                  instance = new ThreadSafeSingleton();
               }
            }
         }

         return instance;
      }
```

```
    }
}
```

Si noti che il controllo `if (instance == null)` viene eseguito due volte: una volta prima dell'acquisizione del blocco e una volta in seguito. Questa implementazione sarebbe ancora thread-safe anche senza il primo controllo nullo. Tuttavia, ciò significherebbe che un blocco sarebbe acquisito *ogni volta che* viene richiesta l'istanza, e ciò causerebbe sofferenza alle prestazioni. Il primo controllo nullo viene aggiunto in modo tale che il blocco non venga acquisito a meno che non sia necessario. Il secondo controllo nullo si assicura che solo il primo thread per acquisire il blocco crei l'istanza. Gli altri thread troveranno l'istanza da compilare e saltare in avanti.

Singleton pigro, sicuro per i thread (usando Pigro)

.Net 4.0 type Lazy garantisce l'inizializzazione degli oggetti thread-safe, quindi questo tipo potrebbe essere usato per creare Singletons.

```
public class LazySingleton
{
    private static readonly Lazy<LazySingleton> _instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    public static LazySingleton Instance
    {
        get { return _instance.Value; }
    }

    private LazySingleton() { }
}
```

Usando `Lazy<T>` ci si assicurerà che l'oggetto sia istanziato solo quando è usato da qualche parte nel codice chiamante.

Un semplice utilizzo sarà come:

```
using System;

public class Program
{
    public static void Main()
    {
        var instance = LazySingleton.Instance;
    }
}
```

Live Demo su .NET Fiddle

Singleton Lazy, thread safe (per .NET 3.5 o versioni precedenti, implementazione alternativa)

Perché in .NET 3.5 e versioni precedenti non hai la classe `Lazy<T>` tu usi il seguente modello:

```
public class Singleton
{
    private Singleton() // prevents public instantiation
    {
    }

    public static Singleton Instance
    {
        get
        {
            return Nested.instance;
        }
    }

    private class Nested
    {
        // Explicit static constructor to tell C# compiler
        // not to mark type as beforefieldinit
        static Nested()
        {
        }

        internal static readonly Singleton instance = new Singleton();
    }
}
```

Questo è ispirato al post sul blog di Jon Skeet .

Poiché la classe `Nested` è nidificata e privata, l'istanza dell'istanza singleton non verrà attivata accedendo ad altri membri della classe `Sigleton` (ad esempio, una proprietà pubblica di sola lettura, ad esempio).

Smaltimento dell'istanza Singleton quando non è più necessaria

La maggior parte degli esempi mostra la `LazySingleton` istanze e il possesso di un oggetto `LazySingleton` fino a quando l'applicazione proprietaria non è terminata, anche se tale oggetto non è più necessario dall'applicazione. Una soluzione è implementare `IDisposable` e impostare l'istanza dell'oggetto su null come segue:

```
public class LazySingleton : IDisposable
{
    private static volatile Lazy<LazySingleton> _instance;
    private static volatile int _instanceCount = 0;
    private bool _alreadyDisposed = false;

public static LazySingleton Instance
{
    get
    {
        if (_instance == null)
            _instance = new Lazy<LazySingleton>(() => new LazySingleton());
        _instanceCount++;
        return _instance.Value;
    }
}

private LazySingleton() { }
```

```
// Public implementation of Dispose pattern callable by consumers.
public void Dispose()
{
    if (--_instanceCount == 0) // No more references to this object.
    {
       Dispose(true);
       GC.SuppressFinalize(this);
    }
}

// Protected implementation of Dispose pattern.
protected virtual void Dispose(bool disposing)
{
    if (_alreadyDisposed) return;

    if (disposing)
    {
        _instance = null; // Allow GC to dispose of this instance.
        // Free any other managed objects here.
    }

    // Free any unmanaged objects here.
    _alreadyDisposed = true;
}
```

Il codice precedente elimina l'istanza prima della chiusura dell'applicazione, ma solo se i consumatori chiamano `Dispose()` sull'oggetto dopo ogni utilizzo. Poiché non vi è alcuna garanzia che ciò accada o un modo per forzarlo, non vi è alcuna garanzia che l'istanza sarà mai smaltita. Ma se questa classe viene utilizzata internamente, è più facile assicurarsi che il metodo `Dispose()` venga chiamato dopo ogni utilizzo. Segue un esempio:

```
public class Program
{
    public static void Main()
    {
        using (var instance = LazySingleton.Instance)
        {
            // Do work with instance
        }
    }
}
```

Si prega di notare che questo esempio **non** è **thread-safe** .

Leggi Implementazione di Singleton online:
https://riptutorial.com/it/csharp/topic/1192/implementazione-di-singleton

Capitolo 76: Importa contatti Google

Osservazioni

I dati dei contatti dell'utente saranno ricevuti in formato JSON, lo estraiamo e finalmente passiamo in rassegna questi dati e otteniamo così i contatti di Google.

Examples

Requisiti

Per importare i contatti di Google (Gmail) nell'applicazione ASP.NET MVC, devi prima scaricare "Impostazione dell'API di Google". Ciò consentirà i seguenti riferimenti:

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
```

Aggiungi questi all'applicazione pertinente.

Codice sorgente nel controller

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Web;
using System.Web.Mvc;

namespace GoogleContactImport.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Import()
        {
            string clientId = "";  // here you need to add your google client id
            string redirectUrl = "http://localhost:1713/Home/AddGoogleContacts"; // here your
redirect action method NOTE: you need to configure same url in google console
            Response.Redirect("https://accounts.google.com/o/oauth2/auth?redirect_uri=" +
```

```
redirectUrl + "&amp&response_type=code&amp&client_id=" + clientId +
"&amp&scope=https://www.google.com/m8/feeds/&amp;approval_prompt=force&amp;access_type=offline");

            return View();
        }

        public ActionResult AddGoogleContacts()
        {
            string code = Request.QueryString["code"];
            if (!string.IsNullOrEmpty(code))
            {
                var contacts = GetAccessToken().ToArray();
                if (contacts.Length > 0)
                {
                  // You will get all contacts here
                    return View("Index",contacts);
                }
                else
                {
                    return RedirectToAction("Index","Home");
                }
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        public List<GmailContacts> GetAccessToken()
        {
            string code = Request.QueryString["code"];
            string google_client_id = ""; //your google client Id
            string google_client_sceret = ""; // your google secret key
            string google_redirect_url = "http://localhost:1713/MyContact/AddGoogleContacts";


            HttpWebRequest webRequest =
(HttpWebRequest)WebRequest.Create("https://accounts.google.com/o/oauth2/token");
            webRequest.Method = "POST";
            string parameters = "code=" + code + "&client_id=" + google_client_id +
"&client_secret=" + google_client_sceret + "&redirect_uri=" + google_redirect_url +
"&grant_type=authorization_code";
            byte[] byteArray = Encoding.UTF8.GetBytes(parameters);
            webRequest.ContentType = "application/x-www-form-urlencoded";
            webRequest.ContentLength = byteArray.Length;
            Stream postStream = webRequest.GetRequestStream();
            // Add the post data to the web request
            postStream.Write(byteArray, 0, byteArray.Length);
            postStream.Close();
            WebResponse response = webRequest.GetResponse();
            postStream = response.GetResponseStream();
            StreamReader reader = new StreamReader(postStream);
            string responseFromServer = reader.ReadToEnd();
            GooglePlusAccessToken serStatus =
JsonConvert.DeserializeObject<GooglePlusAccessToken>(responseFromServer);
            /*End*/
            return GetContacts(serStatus);
        }

        public List<GmailContacts> GetContacts(GooglePlusAccessToken serStatus)
        {
            string google_client_id = "";  //client id
```

```
            string google_client_sceret = ""; //secret key
            /*Get Google Contacts From Access Token and Refresh Token*/
            // string refreshToken = serStatus.refresh_token;
            string accessToken = serStatus.access_token;
            string scopes = "https://www.google.com/m8/feeds/contacts/default/full/";
            OAuth2Parameters oAuthparameters = new OAuth2Parameters()
            {
                ClientId = google_client_id,
                ClientSecret = google_client_sceret,
                RedirectUri = "http://localhost:1713/Home/AddGoogleContacts",
                Scope = scopes,
                AccessToken = accessToken,
                //  RefreshToken = refreshToken
            };

            RequestSettings settings = new RequestSettings("App Name", oAuthparameters);
            ContactsRequest cr = new ContactsRequest(settings);
            ContactsQuery query = new
ContactsQuery(ContactsQuery.CreateContactsUri("default"));
            query.NumberToRetrieve = 5000;
            Feed<Contact> ContactList = cr.GetContacts();

            List<GmailContacts> olist = new List<GmailContacts>();
            foreach (Contact contact in ContactList.Entries)
            {
                foreach (EMail email in contact.Emails)
                {
                    GmailContacts gc = new GmailContacts();
                    gc.EmailID = email.Address;
                    var a = contact.Name.FullName;
                    olist.Add(gc);
                }
            }
            return olist;
        }

        public class GmailContacts
        {
            public string EmailID
            {
                get { return _EmailID; }
                set { _EmailID = value; }
            }
            private string _EmailID;
        }

        public class GooglePlusAccessToken
        {

            public GooglePlusAccessToken()
            { }


            public string access_token
            {
                get { return _access_token; }
                set { _access_token = value; }
            }
            private string _access_token;

            public string token_type
```

```
            {
                get { return _token_type; }
                set { _token_type = value; }
            }
            private string _token_type;

            public string expires_in
            {
                get { return _expires_in; }
                set { _expires_in = value; }
            }
            private string _expires_in;

        }
    }
}
```

Codice sorgente nella vista

L'unico metodo di azione che devi aggiungere è aggiungere un link di azione presente sotto

```
<a href='@Url.Action("Import", "Home")'>Import Google Contacts</a>
```

Leggi Importa contatti Google online: https://riptutorial.com/it/csharp/topic/6744/importa-contatti-google

Capitolo 77: indicizzatore

Sintassi

- public ReturnType this [IndexType index] {ottieni {...} set {...}}

Osservazioni

Indexer consente la sintassi di tipo array per accedere a una proprietà di un oggetto con un indice.

- Può essere utilizzato su una classe, una struttura o un'interfaccia.
- Può essere sovraccaricato
- Può usare più parametri.
- Può essere utilizzato per accedere e impostare valori.
- Può usare qualsiasi tipo per il suo indice.

Examples

Un semplice indicizzatore

```
class Foo
{
    private string[] cities = new[] { "Paris", "London", "Berlin" };

    public string this[int index]
    {
        get {
            return cities[index];
        }
        set {
            cities[index] = value;
        }
    }
}
```

Uso:

```
    var foo = new Foo();

    // access a value
    string berlin = foo[2];

    // assign a value
    foo[0] = "Rome";
```

Visualizza la demo

Indicizzatore con 2 argomenti e interfaccia

```
interface ITable {
    // an indexer can be declared in an interface
    object this[int x, int y] { get; set; }
}

class DataTable : ITable
{
    private object[,] cells = new object[10, 10];

    /// <summary>
    /// implementation of the indexer declared in the interface
    /// </summary>
    /// <param name="x">X-Index</param>
    /// <param name="y">Y-Index</param>
    /// <returns>Content of this cell</returns>
    public object this[int x, int y]
    {
        get
        {
            return cells[x, y];
        }
        set
        {
            cells[x, y] = value;
        }
    }
}
```

Sovraccarico dell'indicizzatore per creare un Sparray

Sovraccaricando l'indicizzatore è possibile creare una classe che ha l'aspetto e la sensazione di un array ma non lo è. Avrà O (1) ottieni e imposta metodi, può accedere ad un elemento nell'indice 100, e tuttavia ha ancora la dimensione degli elementi al suo interno. La classe SparseArray

```
class SparseArray
    {
        Dictionary<int, string> array = new Dictionary<int, string>();

        public string this[int i]
        {
            get
            {
                if(!array.ContainsKey(i))
                {
                    return null;
                }
                return array[i];
            }
            set
            {
                if(!array.ContainsKey(i))
                    array.Add(i, value);
            }
        }
    }
```

Leggi indicizzatore online: https://riptutorial.com/it/csharp/topic/1660/indicizzatore

Capitolo 78: Iniezione di dipendenza

Osservazioni

La definizione di Wikipedia di iniezione di dipendenza è:

Nell'ingegneria del software, l'iniezione di dipendenza è un modello di progettazione software che implementa l'inversione del controllo per la risoluzione delle dipendenze. Una dipendenza è un oggetto che può essere utilizzato (un servizio). Un'iniezione è il passaggio di una dipendenza a un oggetto dipendente (un client) che lo userebbe.

** **Questo sito offre una risposta alla domanda Come spiegare l'iniezione di dipendenza a un bambino di 5 anni. La risposta più votata, fornita da John Munsch, fornisce un'analogia sorprendentemente precisa rivolta al (immaginario) inquisitore quinquennale: quando vai a prendere le cose dal frigorifero per conto tuo, puoi causare problemi. Potresti lasciare la porta aperta, potresti ottenere qualcosa che mamma o papà non vogliono che tu abbia. Potresti anche cercare qualcosa che non abbiamo nemmeno o che è scaduto. Quello che dovresti fare è affermare un bisogno, "Ho bisogno di qualcosa da bere a pranzo", e poi ci assicureremo che tu abbia qualcosa quando ti siedi per mangiare. Cosa significa in termini di sviluppo di software orientato agli oggetti è questo: le classi collaborative (i bambini di cinque anni) dovrebbero fare affidamento sull'infrastruttura (i genitori) per fornire**

** Questo codice utilizza MEF per caricare dinamicamente la dll e risolvere le dipendenze. La dipendenza di ILogger è risolta da MEF e iniettata nella classe utente. La classe utente non riceve mai l'implementazione concreta di ILogger e non ha idea di quale tipo di registratore utilizzi. **

Examples

Iniezione delle dipendenze con MEF

```
public interface ILogger
{
    void Log(string message);
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "Console")]
public class ConsoleLogger:ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "File")]
public class FileLogger:ILogger
{
```

```
    public void Log(string message)
    {
        //Write the message to file
    }
}

public class User
{
    private readonly ILogger logger;
    public User(ILogger logger)
    {
        this.logger = logger;
    }
    public void LogUser(string message)
    {
        logger.Log(message)  ;
    }
}

public interface ILoggerMetaData
{
    string Name { get; }
}

internal class Program
{
    private CompositionContainer _container;

    [ImportMany]
    private IEnumerable<Lazy<ILogger, ILoggerMetaData>> _loggers;

    private static void Main()
    {
        ComposeLoggers();
        Lazy<ILogger, ILoggerMetaData> loggerNameAndLoggerMapping = _ loggers.First((n) =>
((n.Metadata.Name.ToUpper() =="Console"));
        ILogger logger= loggerNameAndLoggerMapping.Value
        var user = new User(logger);
        user.LogUser("user name");
    }

    private void ComposeLoggers()
    {
        //An aggregate catalog that combines multiple catalogs
        var catalog = new AggregateCatalog();
        string loggersDllDirectory =Path.Combine(Utilities.GetApplicationDirectory(),
"Loggers");
        if (!Directory.Exists(loggersDllDirectory ))
        {
            Directory.CreateDirectory(loggersDllDirectory );
        }
        //Adds all the parts found in the same assembly as the PluginManager class
        catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));
        catalog.Catalogs.Add(new DirectoryCatalog(loggersDllDirectory ));

        //Create the CompositionContainer with the parts in the catalog
        _container = new CompositionContainer(catalog);

        //Fill the imports of this object
        try
        {
```

```
            this._container.ComposeParts(this);
        }
        catch (CompositionException compositionException)
        {
            throw new CompositionException(compositionException.Message);
        }
    }
}
```

Dipendenza Injection C # e ASP.NET con Unity

Innanzitutto perché dovremmo usare l'iniezione di depedency nel nostro codice? Vogliamo disaccoppiare altri componenti da altre classi nel nostro programma. Ad esempio abbiamo AnimalController di classe che hanno un codice come questo:

```
public class AnimalController()
{
    private SantaAndHisReindeer _SantaAndHisReindeer = new SantaAndHisReindeer();

    public AnimalController(){
            Console.WriteLine("");
    }
}
```

Guardiamo questo codice e pensiamo che tutto sia a posto, ma ora il nostro AnimalController dipende dall'oggetto _SantaAndHisReindeer. Automaticamente il mio controller non è adatto ai test e la riusabilità del mio codice sarà molto difficile.

Ottima spiegazione del motivo per cui dovremmo usare Injection e interfacce qui .

Se vogliamo che Unity gestisca DI, la strada per raggiungerla è molto semplice :) Con NuGet (gestore pacchetti) possiamo facilmente importare unità nel nostro codice.

> in Visual Studio Tools -> NuGet Package Manager -> Gestisci pacchetti per soluzione -> in input di ricerca scrivi unità -> scegli il nostro progetto-> fai clic su Installa

Ora verranno creati due file con commenti interessanti.

> nella cartella App-Data UnityConfig.cs e UnityMvcActivator.cs
>
> UnityConfig - nel metodo RegisterTypes, possiamo vedere il tipo che verrà inserito nei nostri costruttori.

```
namespace Vegan.WebUi.App_Start
{

public class UnityConfig
{
    #region Unity Container
    private static Lazy<IUnityContainer> container = new Lazy<IUnityContainer>(() =>
    {
        var container = new UnityContainer();
        RegisterTypes(container);
```

```
        return container;
    });

    /// <summary>
    /// Gets the configured Unity container.
    /// </summary>
    public static IUnityContainer GetConfiguredContainer()
    {
        return container.Value;
    }
    #endregion

    /// <summary>Registers the type mappings with the Unity container.</summary>
    /// <param name="container">The unity container to configure.</param>
    /// <remarks>There is no need to register concrete types such as controllers or API
controllers (unless you want to
    /// change the defaults), as Unity allows resolving a concrete type even if it was not
previously registered.</remarks>
    public static void RegisterTypes(IUnityContainer container)
    {
        // NOTE: To load from web.config uncomment the line below. Make sure to add a
Microsoft.Practices.Unity.Configuration to the using statements.
        // container.LoadConfiguration();

        // TODO: Register your types here
        // container.RegisterType<IProductRepository, ProductRepository>();

        container.RegisterType<ISanta, SantaAndHisReindeer>();

     }
 }
}
```

UnityMvcActivator -> anche con dei bei commenti che dicono che questa classe integra Unity con ASP.NET MVC

```
using System.Linq;
using System.Web.Mvc;
using Microsoft.Practices.Unity.Mvc;

[assembly:
WebActivatorEx.PreApplicationStartMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Start")]
[assembly:
WebActivatorEx.ApplicationShutdownMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Shutdown")]

namespace Vegan.WebUi.App_Start
{
/// <summary>Provides the bootstrapping for integrating Unity with ASP.NET MVC.</summary>
public static class UnityWebActivator
{
    /// <summary>Integrates Unity when the application starts.</summary>
    public static void Start()
    {
        var container = UnityConfig.GetConfiguredContainer();


FilterProviders.Providers.Remove(FilterProviders.Providers.OfType<FilterAttributeFilterProvider>().Firs
```

```
        FilterProviders.Providers.Add(new UnityFilterAttributeFilterProvider(container));

        DependencyResolver.SetResolver(new UnityDependencyResolver(container));

        // TODO: Uncomment if you want to use PerRequestLifetimeManager
        //
Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility.RegisterModule(typeof(UnityPerRe

    }

    /// <summary>Disposes the Unity container when the application is shut down.</summary>
    public static void Shutdown()
    {
        var container = UnityConfig.GetConfiguredContainer();
        container.Dispose();
    }
}
}
```

Ora possiamo disaccoppiare il nostro controller dalla classe SantAndHisReindeer :)

```
 public class AnimalController()
    {
        private readonly SantaAndHisReindeer _SantaAndHisReindeer;

        public AnimalController(SantaAndHisReindeer SantaAndHisReindeer){

                _SantAndHisReindeer = SantaAndHisReindeer;
        }
    }
```

C'è un'ultima cosa che dobbiamo fare prima di eseguire la nostra applicazione.

In Global.asax.cs dobbiamo aggiungere una nuova riga: UnityWebActivator.Start () che verrà avviato, configurare Unity e registrare i nostri tipi.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using Vegan.WebUi.App_Start;

namespace Vegan.WebUi
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            UnityWebActivator.Start();
        }
    }
```

```
}
```

Leggi Iniezione di dipendenza online: https://riptutorial.com/it/csharp/topic/5766/iniezione-di-dipendenza

Capitolo 79: Inizializzatori di oggetti

Sintassi

- SomeClass sc = new SomeClass {Property1 = value1, Property2 = value2, ...};
- SomeClass sc = new SomeClass (param1, param2, ...) {Proprietà1 = valore1, Proprietà2 = valore2, ...}

Osservazioni

Le parentesi del costruttore possono essere omesse solo se il tipo istanziato ha un costruttore predefinito (senza parametri) disponibile.

Examples

Uso semplice

Gli inizializzatori degli oggetti sono utili quando è necessario creare un oggetto e impostare subito un paio di proprietà, ma i costruttori disponibili non sono sufficienti. Di 'che hai una lezione

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }

    // the rest of class definition
}
```

Per inizializzare una nuova istanza della classe con un inizializzatore:

```
Book theBook = new Book { Title = "Don Quixote", Author = "Miguel de Cervantes" };
```

Questo è equivalente a

```
Book theBook = new Book();
theBook.Title = "Don Quixote";
theBook.Author = "Miguel de Cervantes";
```

Utilizzo con tipi anonimi

Gli inizializzatori degli oggetti sono l'unico modo per inizializzare i tipi anonimi, che sono tipi generati dal compilatore.

```
var album = new { Band = "Beatles", Title = "Abbey Road" };
```

Per questo motivo gli inizializzatori di oggetti sono ampiamente utilizzati nelle query di selezione

LINQ, poiché forniscono un modo conveniente per specificare quali parti di un oggetto interrogato sono interessati.

```
var albumTitles = from a in albums
                  select new
                  {
                     Title = a.Title,
                     Artist = a.Band
                  };
```

Utilizzo con costruttori non predefiniti

È possibile combinare gli inizializzatori di oggetti con i costruttori per inizializzare i tipi, se necessario. Prendiamo ad esempio una classe definita come tale:

```
public class Book {
    public string Title { get; set; }
    public string Author { get; set; }

    public Book(int id) {
        //do things
    }

    // the rest of class definition
}

var someBook = new Book(16) { Title = "Don Quixote", Author = "Miguel de Cervantes" }
```

Ciò prima istanzia un costruttore `Book` with the `Book(int)` , quindi imposta ogni proprietà nell'inizializzatore. È equivalente a:

```
var someBook = new Book(16);
someBook.Title = "Don Quixote";
someBook.Author = "Miguel de Cervantes";
```

Leggi Inizializzatori di oggetti online: https://riptutorial.com/it/csharp/topic/738/inizializzatori-di-oggetti

Capitolo 80: Inizializzatori di raccolta

Osservazioni

L'unico requisito per l'inizializzazione di un oggetto utilizzando questo zucchero sintattico è che il tipo implementa `System.Collections.IEnumerable` e il metodo `Add` . Sebbene lo chiamiamo inizializzatore di una raccolta, l'oggetto *non* deve essere una raccolta.

Examples

Inizializzatori di raccolta

Inizializza un tipo di raccolta con valori:

```
var stringList = new List<string>
{
    "foo",
    "bar",
};
```

Gli inizializzatori della raccolta sono zucchero sintattico per le chiamate `Add()` . Sopra il codice è equivalente a:

```
var temp = new List<string>();
temp.Add("foo");
temp.Add("bar");
var stringList = temp;
```

Si noti che l'intializzazione viene eseguita atomicamente utilizzando una variabile temporanea, per evitare condizioni di competizione.

Per i tipi che offrono più parametri nel loro metodo `Add()` , racchiudi gli argomenti separati da virgola in parentesi graffe:

```
var numberDictionary = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
};
```

Questo è equivalente a:

```
var temp = new Dictionary<int, string>();
temp.Add(1, "One");
temp.Add(2, "Two");
var numberDictionarynumberDictionary = temp;
```

Inizializzatori dell'indice C # 6

A partire da C # 6, le raccolte con indicizzatori possono essere inizializzate specificando l'indice da assegnare tra parentesi quadre, seguito da un segno di uguale, seguito dal valore da assegnare.

Inizializzazione del dizionario

Un esempio di questa sintassi utilizzando un dizionario:

```
var dict = new Dictionary<string, int>
{
    ["key1"] = 1,
    ["key2"] = 50
};
```

Questo è equivalente a:

```
var dict = new Dictionary<string, int>();
dict["key1"] = 1;
dict["key2"] = 50
```

La sintassi di inizializzazione della raccolta per eseguire questa operazione prima di C # 6 era:

```
var dict = new Dictionary<string, int>
{
    { "key1", 1 },
    { "key2", 50 }
};
```

Quale corrisponderebbe a:

```
var dict = new Dictionary<string, int>();
dict.Add("key1", 1);
dict.Add("key2", 50);
```

Quindi c'è una differenza significativa nella funzionalità, poiché la nuova sintassi usa l' *indicizzatore* dell'oggetto inizializzato per assegnare valori invece di usare il suo metodo `Add()` . Ciò significa che la nuova sintassi richiede solo un indicizzatore pubblicamente disponibile e funziona per qualsiasi oggetto che ne ha uno.

```
public class IndexableClass
{
    public int this[int index]
    {
        set
        {
            Console.WriteLine("{0} was assigned to index {1}", value, index);
        }
    }
```

```
}

var foo = new IndexableClass
{
    [0] = 10,
    [1] = 20
}
```

Ciò produrrebbe:

```
10 was assigned to index 0
20 was assigned to index 1
```

Inizializzatori di raccolta in classi personalizzate

Per creare un inizializzatore della raccolta del supporto classe, deve implementare l'interfaccia `IEnumerable` e avere almeno un metodo `Add` . Dal momento che C # 6, qualsiasi raccolta attuare `IEnumerable` può essere esteso con personalizzati `Add` metodi che utilizzano metodi di estensione.

```
class Program
{
    static void Main()
    {
        var col = new MyCollection {
            "foo",
            { "bar", 3 },
            "baz",
            123.45d,
        };
    }
}

class MyCollection : IEnumerable
{
    private IList list = new ArrayList();

    public void Add(string item)
    {
        list.Add(item)
    }

    public void Add(string item, int count)
    {
        for(int i=0;i< count;i++) {
            list.Add(item);
        }
    }

    public IEnumerator GetEnumerator()
    {
        return list.GetEnumerator();
    }
}

static class MyCollectionExtensions
{
    public static void Add(this MyCollection @this, double value) =>
        @this.Add(value.ToString());
```

```
}
```

Inizializzatori di raccolta con array di parametri

È possibile combinare parametri normali e array di parametri:

```
public class LotteryTicket : IEnumerable{
    public int[] LuckyNumbers;
    public string UserName;

    public void Add(string userName, params int[] luckyNumbers){
        UserName = userName;
        Lottery = luckyNumbers;
    }
}
```

Questa sintassi è ora possibile:

```
var Tickets = new List<LotteryTicket>{
    {"Mr Cool"  , 35663, 35732, 12312, 75685},
    {"Bruce"    , 26874, 66677, 24546, 36483, 46768, 24632, 24527},
    {"John Cena", 25446, 83356, 65536, 23783, 24567, 89337}
}
```

Utilizzo di inizializzatore di raccolta nell'inizializzatore dell'oggetto

```
public class Tag
{
    public IList<string> Synonyms { get; set; }
}
```

`Synonyms` è una proprietà del tipo di raccolta. Quando l'oggetto `Tag` viene creato utilizzando la sintassi di inizializzazione dell'oggetto, i `Synonyms` possono anche essere inizializzati con la sintassi di inizializzazione della raccolta:

```
Tag t = new Tag
{
    Synonyms = new List<string> {"c#", "c-sharp"}
};
```

La proprietà della raccolta può essere in sola lettura e supporta ancora la sintassi di inizializzazione della raccolta. Considera questo esempio modificato (la proprietà `Synonyms` ora ha un setter privato):

```
public class Tag
{
    public Tag()
    {
        Synonyms = new List<string>();
    }

    public IList<string> Synonyms { get; private set; }
```

```
}
```

Un nuovo oggetto `Tag` può essere creato in questo modo:

```
Tag t = new Tag
{
    Synonyms = {"c#", "c-sharp"}
};
```

Funziona perché gli inizializzatori della raccolta sono solo zucchero sintattico sulle chiamate a `Add()` . Non è stato creato alcun nuovo elenco qui, il compilatore sta solo generando chiamate a `Add()` sull'oggetto che sta uscendo.

Leggi Inizializzatori di raccolta online: https://riptutorial.com/it/csharp/topic/21/inizializzatori-di-raccolta

Capitolo 81: Inizializzazione delle proprietà

Osservazioni

Al momento di decidere come creare una proprietà, iniziare con una proprietà auto-implementata per semplicità e brevità.

Passa a una proprietà con un campo di supporto solo quando le circostanze lo richiedono. Se hai bisogno di altre manipolazioni oltre un semplice set e ottieni, potresti dover introdurre un backing field.

Examples

C # 6.0: inizializzazione di una proprietà implementata automaticamente

Crea una proprietà con getter e / o setter e inizializza tutto in una riga:

```
public string Foobar { get; set; } = "xyz";
```

Inizializzazione della proprietà con un campo di supporto

```
public string Foobar {
    get { return _foobar; }
    set { _foobar = value; }
}
private string _foobar = "xyz";
```

Inizializzazione della proprietà in Costruttore

```
class Example
{
    public string Foobar { get; set; }
    public List<string> Names { get; set; }
    public Example()
    {
        Foobar = "xyz";
        Names = new List<string>(){"carrot","fox","ball"};
    }
}
```

Inizializzazione della proprietà durante l'istanziazione dell'oggetto

Le proprietà possono essere impostate quando un oggetto viene istanziato.

```
var redCar = new Car
{
    Wheels = 2,
```

```
    Year = 2016,
    Color = Color.Red
};
```

Leggi Inizializzazione delle proprietà online:
https://riptutorial.com/it/csharp/topic/82/inizializzazione-delle-proprieta

Capitolo 82: interfacce

Examples

Implementazione di un'interfaccia

Un'interfaccia viene utilizzata per forzare la presenza di un metodo in qualsiasi classe che lo "implementa". L'interfaccia è definita con l' `interface` per le parole chiave e una classe può "implementarla" aggiungendo `: InterfaceName` dopo il nome della classe. Una classe può implementare più interfacce separando ciascuna interfaccia con una virgola.
`: InterfaceName, ISecondInterface`

```
public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : INoiseMaker
{
    public string MakeNoise()
    {
        return "Nyan";
    }
}

public class Dog : INoiseMaker
{
    public string MakeNoise()
    {
        return "Woof";
    }
}
```

Poiché implementano `INoiseMaker` , sia il `cat` che il `dog` devono includere il metodo `string MakeNoise()` e non potranno compilare senza di esso.

Implementazione di più interfacce

```
public interface IAnimal
{
    string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
```

```
    }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

Implementazione esplicita dell'interfaccia

L'implementazione esplicita dell'interfaccia è necessaria quando si implementano più interfacce che definiscono un metodo comune, ma sono necessarie implementazioni diverse a seconda dell'interfaccia utilizzata per chiamare il metodo (si noti che non sono necessarie implementazioni esplicite se più interfacce condividono lo stesso metodo e è possibile un'implementazione comune).

```
interface IChauffeur
{
    string Drive();
}

interface IGolfPlayer
{
    string Drive();
}

class GolfingChauffeur : IChauffeur, IGolfPlayer
{
    public string Drive()
    {
        return "Vroom!";
    }

    string IGolfPlayer.Drive()
    {
        return "Took a swing...";
    }
}


GolfingChauffeur obj = new GolfingChauffeur();
IChauffeur chauffeur = obj;
IGolfPlayer golfer = obj;

Console.WriteLine(obj.Drive()); // Vroom!
Console.WriteLine(chauffeur.Drive()); // Vroom!
Console.WriteLine(golfer.Drive()); // Took a swing...
```

L'implementazione non può essere richiamata da nessun'altra parte se non utilizzando l'interfaccia:

```
public class Golfer : IGolfPlayer
{
    string IGolfPlayer.Drive()
```

```
    {
        return "Swinging hard...";
    }
    public void Swing()
    {
        Drive(); // Compiler error: No such method
    }
}
```

A causa di ciò, può essere vantaggioso inserire un codice di implementazione complesso di un'interfaccia esplicitamente implementata in un metodo privato separato.

Un'implementazione esplicita dell'interfaccia può naturalmente essere utilizzata solo per i metodi effettivamente esistenti per quell'interfaccia:

```
public class ProGolfer : IGolfPlayer
{
    string IGolfPlayer.Swear() // Error
    {
        return "The ball is in the pit";
    }
}
```

Allo stesso modo, l'utilizzo di un'implementazione esplicita dell'interfaccia senza dichiarare che l'interfaccia sulla classe causa anche un errore.

Suggerimento:

Implementare le interfacce in modo esplicito può anche essere usato per evitare il dead code. Quando un metodo non è più necessario e viene rimosso dall'interfaccia, il compilatore si lamenterà di ogni implementazione ancora esistente.

Nota:

I programmatori si aspettano che il contratto sia lo stesso indipendentemente dal contesto del tipo e l'implementazione esplicita non dovrebbe esporre un comportamento diverso quando viene chiamata. Quindi, a differenza dell'esempio sopra, `IGolfPlayer.Drive` e `Drive` dovrebbero fare la stessa cosa quando possibile.

Perché usiamo le interfacce

Un'interfaccia è una definizione di contratto tra l'utente dell'interfaccia e la classe che lo implementa. Un modo per pensare a un'interfaccia è come una dichiarazione che un oggetto può eseguire determinate funzioni.

Diciamo che definiamo un'interfaccia `IShape` per rappresentare diversi tipi di forme, ci aspettiamo che una forma abbia un'area, quindi definiremo un metodo per forzare le implementazioni dell'interfaccia a restituire la loro area:

```
public interface IShape
{
    double ComputeArea();
}
```

Vediamo che abbiamo le seguenti due forme: un `Rectangle` e un `Circle`

```
public class Rectangle : IShape
{
    private double length;
    private double width;

    public Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
    }

    public double ComputeArea()
    {
        return length * width;
    }
}

public class Circle : IShape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public double ComputeArea()
    {
        return Math.Pow(radius, 2.0) * Math.PI;
    }
}
```

Ognuno di loro ha la propria definizione della propria area, ma entrambi sono forme. Quindi è logico vederli come `IShape` nel nostro programma:

```
private static void Main(string[] args)
{
    var shapes = new List<IShape>() { new Rectangle(5, 10), new Circle(5) };
    ComputeArea(shapes);

    Console.ReadKey();
}

private static void ComputeArea(IEnumerable<IShape> shapes)
{
    foreach (shape in shapes)
    {
        Console.WriteLine("Area: {0:N}, shape.ComputeArea());
    }
}
```

```
// Output:
// Area : 50.00
// Area : 78.54
```

Nozioni di base sull'interfaccia

La funzione di un'interfaccia nota come "contratto" di funzionalità. Significa che dichiara proprietà e metodi ma non li implementa.

Quindi, a differenza delle interfacce di classi:

- Non può essere istanziato
- Non può avere alcuna funzionalità
- Può contenere solo metodi * *(Proprietà ed Eventi sono metodi internamente)*
- L'ereditarietà di un'interfaccia si chiama "Implementazione"
- È possibile ereditare da 1 classe, ma è possibile "Implementare" più interfacce

```
public interface ICanDoThis{
    void TheThingICanDo();
    int SomeValueProperty { get; set; }
}
```

Cose da notare:

- Il prefisso "I" è una convenzione di denominazione utilizzata per le interfacce.
- Il corpo della funzione viene sostituito con un punto e virgola ";".
- Le proprietà sono anche permesse perché internamente sono anche metodi

```
public class MyClass : ICanDoThis {
    public void TheThingICanDo(){
        // do the thing
    }

    public int SomeValueProperty { get; set; }
    public int SomeValueNotImplemtingAnything { get; set; }
}
```

.

```
ICanDoThis obj = new MyClass();

// ok
obj.TheThingICanDo();

// ok
obj.SomeValueProperty = 5;

// Error, this member doesn't exist in the interface
obj.SomeValueNotImplemtingAnything = 5;

// in order to access the property in the class you must "down cast" it
((MyClass)obj).SomeValueNotImplemtingAnything = 5; // ok
```

Ciò è particolarmente utile quando si lavora con framework UI come WinForms o WPF perché è obbligatorio ereditare da una classe base per creare il controllo utente e si perde la possibilità di creare astrazioni su diversi tipi di controllo. Un esempio? In arrivo:

```
public class MyTextBlock : TextBlock {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button {
    public void SetText(string str){
        this.Content = str;
    }
}
```

Il problema proposto è che entrambi contengono un concetto di "Testo" ma i nomi delle proprietà differiscono. E non puoi creare creare una *classe base astratta* perché hanno un'eredità obbligatoria per 2 classi diverse. Un'interfaccia può alleggerirlo

```
public interface ITextControl{
    void SetText(string str);
}

public class MyTextBlock : TextBlock, ITextControl {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button, ITextControl {
    public void SetText(string str){
        this.Content = str;
    }

    public int Clicks { get; set; }
}
```

Ora MyButton e MyTextBlock sono intercambiabili.

```
var controls = new List<ITextControls>{
    new MyTextBlock(),
    new MyButton()
};

foreach(var ctrl in controls){
    ctrl.SetText("This text will be applied to both controls despite them being different");


    // Compiler Error, no such member in interface
    ctrl.Clicks = 0;

    // Runtime Error because 1 class is in fact not a button which makes this cast invalid
    ((MyButton)ctrl).Clicks = 0;


    /* the solution is to check the type first.
```

```
    This is usually considered bad practice since
    it's a symptom of poor abstraction */
    var button = ctrl as MyButton;
    if(button != null)
        button.Clicks = 0; // no errors


}
```

"Nascondere" i membri con implementazione esplicita

Non lo odi quando le interfacce inquinano la tua classe con troppi membri di cui non ti importa nemmeno? Bene, ho una soluzione! Implementazioni esplicite

```
public interface IMessageService {
    void OnMessageRecieve();
    void SendMessage();
    string Result { get; set; }
    int Encoding { get; set; }
    // yadda yadda
}
```

Normalmente dovresti implementare la classe in questo modo.

```
public class MyObjectWithMessages : IMessageService {
     public void OnMessageRecieve(){

     }

     public void SendMessage(){

     }

     public string Result { get; set; }
     public int Encoding { get; set; }
}
```

Ogni membro è pubblico.

```
var obj = new MyObjectWithMessages();

// why would i want to call this function?
obj.OnMessageRecieve();
```

Risposta: non lo so Quindi nessuno dei due dovrebbe essere dichiarato pubblico, ma semplicemente dichiarare i membri come privati farà sì che il compilatore lanci un errore

La soluzione è usare un'implementazione esplicita:

```
public class MyObjectWithMessages : IMessageService{
    void IMessageService.OnMessageRecieve() {

    }
```

```
    void IMessageService.SendMessage() {

    }

    string IMessageService.Result { get; set; }
    int IMessageService.Encoding { get; set; }
}
```

Quindi ora hai implementato i membri come richiesto e non espongono alcun membro come pubblico.

```
var obj = new MyObjectWithMessages();

/* error member does not exist on type MyObjectWithMessages.
 * We've succesfully made it "private" */
obj.OnMessageRecieve();
```

Se si desidera ancora accedere seriamente al membro anche se è esplicitamente implementato tutto ciò che si deve fare è gettare l'oggetto nell'interfaccia e si è pronti per andare.

```
((IMessageService)obj).OnMessageRecieve();
```

IComparable come esempio di implementazione di un'interfaccia

Le interfacce possono sembrare astratte fino a quando non le appari in pratica. I `IComparable<T>` `IComparable` e `IComparable<T>` sono ottimi esempi del perché le interfacce possono essere utili a noi.

Diciamo che in un programma per un negozio online, abbiamo una varietà di oggetti che puoi comprare. Ogni articolo ha un nome, un numero identificativo e un prezzo.

```
public class Item {

    public string name; // though public variables are generally bad practice,
    public int idNumber; // to keep this example simple we will use them instead
    public decimal price; // of a property.

    // body omitted for brevity

}
```

Abbiamo i nostri `Item` s memorizzati all'interno di un `List<Item>` e nel nostro programma da qualche parte, vogliamo ordinare il nostro elenco per numero ID dal più piccolo al più grande. Invece di scrivere il nostro algoritmo di ordinamento, possiamo invece usare il metodo `Sort()` che `List<T>` ha già. Tuttavia, poiché la nostra classe `Item` è in questo momento, non c'è modo per `List<T>` di capire quale ordine ordinare l'elenco. Qui è dove entra in `IComparable` interfaccia `IComparable` .

Per implementare correttamente il metodo `CompareTo` , `CompareTo` dovrebbe restituire un numero positivo se il parametro è "minore di" quello corrente, zero se sono uguali e un numero negativo se il parametro è "maggiore di".

```
Item apple = new Item();
```

```
apple.idNumber = 15;
Item banana = new Item();
banana.idNumber = 4;
Item cow = new Item();
cow.idNumber = 15;
Item diamond = new Item();
diamond.idNumber = 18;

Console.WriteLine(apple.CompareTo(banana)); // 11
Console.WriteLine(apple.CompareTo(cow)); // 0
Console.WriteLine(apple.CompareTo(diamond)); // -3
```

Ecco l'esempio `Item` implementazione 's dell'interfaccia:

```
public class Item : IComparable<Item> {

    private string name;
    private int idNumber;
    private decimal price;

    public int CompareTo(Item otherItem) {

        return (this.idNumber - otherItem.idNumber);

    }

    // rest of code omitted for brevity

}
```

A livello di superficie, il metodo `CompareTo` nel nostro articolo restituisce semplicemente la differenza nei loro numeri ID, ma cosa fa in pratica quanto sopra?

Ora, quando chiamiamo `Sort()` su un oggetto `List<Item>` , `List` chiamerà automaticamente il metodo `CompareTo Item` quando deve determinare in che ordine inserire gli oggetti. Inoltre, oltre a `List<T>` , qualsiasi altro oggetto che ha bisogno della capacità di confrontare due oggetti funzionerà con l' `Item` perché abbiamo definito la capacità di confrontare due `Item` diversi tra loro.

Leggi interfacce online: https://riptutorial.com/it/csharp/topic/2208/interfacce

Capitolo 83: Interfaccia IDisposable

Osservazioni

- Spetta ai client della classe che implementano `IDisposable` per assicurarsi che chiamino il metodo `Dispose` quando hanno finito di utilizzare l'oggetto. Non c'è nulla nel CLR che cerca direttamente oggetti per un metodo `Dispose` da richiamare.
- Non è necessario implementare un finalizzatore se l'oggetto contiene solo risorse gestite. Assicurati di chiamare `Dispose` su tutti gli oggetti che la tua classe usa quando implementi il tuo metodo di `Dispose` .
- Si consiglia di rendere la classe sicura contro più chiamate a `Dispose` , sebbene dovrebbe idealmente essere chiamata una sola volta. Questo può essere ottenuto aggiungendo una variabile `private bool` alla classe e impostando il valore su `true` quando è stato eseguito il metodo `Dispose` .

Examples

In una classe che contiene solo risorse gestite

Le risorse gestite sono risorse che il garbage collector del runtime è a conoscenza e sotto il controllo di. Esistono molte classi disponibili nel BCL, ad esempio, come `SqlConnection` che è una classe wrapper per una risorsa non gestita. Queste classi implementano già l'interfaccia `IDisposable` - `IDisposable` dal tuo codice per ripulirle quando hai finito.

Non è necessario implementare un finalizzatore se la tua classe contiene solo risorse gestite.

```
public class ObjectWithManagedResourcesOnly : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();

    public void Dispose()
    {
        sqlConnection.Dispose();
    }
}
```

In una classe con risorse gestite e non gestite

È importante lasciare che la finalizzazione ignori le risorse gestite. Il finalizzatore viene eseguito su un altro thread: è possibile che gli oggetti gestiti non esistano più al momento dell'esecuzione del finalizzatore. L'implementazione di un metodo `Dispose(bool)` protetto `Dispose(bool)` è una pratica comune per garantire che le risorse gestite non abbiano il loro metodo `Dispose` chiamato da un finalizzatore.

```
public class ManagedAndUnmanagedObject : IDisposable
```

```
{
    private SqlConnection sqlConnection = new SqlConnection();
    private UnmanagedHandle unmanagedHandle = Win32.SomeUnmanagedResource();
    private bool disposed;

    public void Dispose()
    {
        Dispose(true); // client called dispose
        GC.SuppressFinalize(this); // tell the GC to not execute the Finalizer
    }

    protected virtual void Dispose(bool disposeManaged)
    {
        if (!disposed)
        {
            if (disposeManaged)
            {
                if (sqlConnection != null)
                {
                    sqlConnection.Dispose();
                }
            }

            unmanagedHandle.Release();

            disposed = true;
        }
    }

    ~ManagedAndUnmanagedObject()
    {
        Dispose(false);
    }
}
```

IDisposable, Dispose

.NET Framework definisce un'interfaccia per i tipi che richiedono un metodo di rimozione:

```
public interface IDisposable
{
  void Dispose();
}
```

`Dispose()` viene utilizzato principalmente per la pulizia delle risorse, come i riferimenti non gestiti. Tuttavia, può anche essere utile forzare lo smaltimento di altre risorse anche se sono gestite. Invece di aspettare che il GC risolva anche la tua connessione al database, puoi assicurarti che sia fatto nella tua implementazione di `Dispose()` .

```
public void Dispose()
{
   if (null != this.CurrentDatabaseConnection)
   {
       this.CurrentDatabaseConnection.Dispose();
       this.CurrentDatabaseConnection = null;
   }
}
```

Quando è necessario accedere direttamente alle risorse non gestite come puntatori non gestiti o risorse win32, creare una classe ereditata da `SafeHandle` e utilizzare le convenzioni / gli strumenti di quella classe per farlo.

In una classe ereditata con risorse gestite

È abbastanza comune che sia possibile creare una classe che implementa `IDisposable` e quindi derivare classi che contengono anche risorse gestite. Si consiglia di contrassegnare il metodo `Dispose` con la parola chiave `virtual` modo che i client abbiano la possibilità di ripulire tutte le risorse che possono possedere.

```
public class Parent : IDisposable
{
    private ManagedResource parentManagedResource = new ManagedResource();

    public virtual void Dispose()
    {
        if (parentManagedResource != null)
        {
            parentManagedResource.Dispose();
        }
    }
}

public class Child : Parent
{
    private ManagedResource childManagedResource = new ManagedResource();

    public override void Dispose()
    {
        if (childManagedResource != null)
        {
            childManagedResource.Dispose();
        }
        //clean up the parent's resources
        base.Dispose();
    }
}
```

usando la parola chiave

Quando un oggetto implementa l'interfaccia `IDisposable` , può essere creato all'interno della sintassi `using` :

```
using (var foo = new Foo())
{
    // do foo stuff
} // when it reaches here foo.Dispose() will get called

public class Foo : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("dispose called");
    }
```

```
}
```

Guarda la demo

`using` è zucchero sintetico per un blocco `try/finally` ; l'utilizzo di cui sopra si tradurrebbe approssimativamente in:

```
{
    var foo = new Foo();
    try
    {
        // do foo stuff
    }
    finally
    {
        if (foo != null)
            ((IDisposable)foo).Dispose();
    }
}
```

Leggi Interfaccia IDisposable online: https://riptutorial.com/it/csharp/topic/1795/interfaccia-idisposable

Capitolo 84: Interfaccia INotifyPropertyChanged

Osservazioni

L'interfaccia `INotifyPropertyChanged` è necessaria ogni volta che è necessario per rendere la classe segnalare le modifiche che si verificano alle sue proprietà. L'interfaccia definisce un singolo evento `PropertyChanged` .

Con XAML Binding l'evento `PropertyChanged` è cablato automaticamente, quindi è necessario implementare l'interfaccia INotifyPropertyChanged sul proprio modello di vista o sulle classi di contesto dati per lavorare con XAML Binding.

Examples

Implementazione di INotifyPropertyChanged in C # 6

L'implementazione di `INotifyPropertyChange` può essere soggetta a errori, poiché l'interfaccia richiede di specificare il nome della proprietà come stringa. Per rendere l'implementazione più solida, è possibile utilizzare un attributo `CallerMemberName` .

```
class C : INotifyPropertyChanged
{
    // backing field
    int offset;
    // property
    public int Offset
    {
        get
        {
            return offset;
        }
        set
        {
            if (offset == value)
                return;
            offset = value;
            RaisePropertyChanged();
        }
    }

    // helper method for raising PropertyChanged event
    void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Se hai diverse classi che implementano `INotifyPropertyChanged` , potresti trovare utile rifattorizzare

l'implementazione dell'interfaccia e il metodo helper alla classe base comune:

```
class NotifyPropertyChangedImpl : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}

class C : NotifyPropertyChangedImpl
{
    int offset;
    public int Offset
    {
        get { return offset; }
        set { if (offset != value) { offset = value; RaisePropertyChanged(); } }
    }
}
```

InotifyPropertyChanged con metodo Set generico

La classe `NotifyPropertyChangedBase` seguito definisce un metodo Set generico che può essere chiamato da qualsiasi tipo derivato.

```
public class NotifyPropertyChangedBase : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    public event PropertyChangedEventHandler PropertyChanged;

    public virtual bool Set<T>(ref T field, T value, [CallerMemberName] string propertyName =
null)
    {
        if (Equals(field, value))
            return false;
        storage = value;
        RaisePropertyChanged(propertyName);
        return true;
    }
}
```

Per utilizzare questo metodo Set generico, è sufficiente creare una classe che derivi da NotifyPropertyChangedBase.

```
public class SomeViewModel : NotifyPropertyChangedBase
{
    private string _foo;
    private int _bar;

    public string Foo
    {
        get { return _foo; }
        set { Set(ref _foo, value); }
```

```
    }

    public int Bar
    {
        get { return _bar; }
        set { Set(ref _bar, value); }
    }
}
```

Come mostrato sopra, puoi chiamare `Set(ref _fieldName, value);` nel setter di una proprietà e genererà automaticamente un evento PropertyChanged se necessario.

È quindi possibile registrarsi all'evento PropertyChanged da un'altra classe che deve gestire le modifiche alle proprietà.

```
public class SomeListener
{
    public SomeListener()
    {
        _vm = new SomeViewModel();
        _vm.PropertyChanged += OnViewModelPropertyChanged;
    }

    private void OnViewModelPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        Console.WriteLine($"Property {e.PropertyName} was changed.");
    }

    private readonly SomeViewModel _vm;

}
```

Leggi Interfaccia INotifyPropertyChanged online:
https://riptutorial.com/it/csharp/topic/2990/interfaccia-inotifypropertychanged

Capitolo 85: Interfaccia IQueryable

Examples

Tradurre una query LINQ in una query SQL

Le `IQueryable` e `IQueryable<T>` consentono agli sviluppatori di tradurre una query LINQ (una query "language-integrated") in un'origine dati specifica, ad esempio un database relazionale. Prendi questa query LINQ scritta in C #:

```
var query = from book in books
            where book.Author == "Stephen King"
            select book;
```

Se la variabile `books` è di un tipo che implementa `IQueryable<Book>` la query precedente viene passata al provider (impostata sulla proprietà `IQueryable.Provider`) sotto forma di un albero di espressioni, una struttura di dati che riflette la struttura del codice .

Il provider può ispezionare l'albero delle espressioni in fase di esecuzione per determinare:

- che esiste un predicato per la proprietà `Author` della classe `Book` ;
- che il metodo di confronto utilizzato è 'uguale' (`==`);
- che il valore che dovrebbe eguagliare è `"Stephen King"` .

Con queste informazioni il provider può tradurre la query C # in una query SQL in fase di runtime e passare quella query a un database relazionale per recuperare solo quei libri che corrispondono al predicato:

```
select *
from Books
where Author = 'Stephen King'
```

Il provider viene chiamato quando la variabile di `query` viene iterata su (`IQueryable` implementa `IEnumerable`).

(Il provider utilizzato in questo esempio richiede alcuni metadati aggiuntivi per sapere quale tabella interrogare e sapere come abbinare le proprietà della classe C # alle colonne della tabella, ma tali metadati sono al di fuori dell'ambito dell'interfaccia `IQueryable` .)

Leggi Interfaccia IQueryable online: https://riptutorial.com/it/csharp/topic/3094/interfaccia-iqueryable

Capitolo 86: interoperabilità

Osservazioni

Lavorare con API Win32 usando C

Windows espone molte funzionalità sotto forma di API Win32. Usando queste API è possibile eseguire operazioni dirette in Windows, il che aumenta le prestazioni della tua applicazione. Source Clicca qui

Windows espone un'ampia gamma di API. Per ottenere informazioni su varie API puoi controllare siti come pinvoke .

Examples

Funzione di importazione da DLL C ++ non gestita

Ecco un esempio di come importare una funzione definita in una DLL C ++ non gestita. Nel codice sorgente C ++ per "myDLL.dll", la funzione `add` è definita:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
{
    return a + b;
}
```

Quindi può essere incluso in un programma C # come segue:

```
class Program
{
    // This line will import the C++ method.
    // The name specified in the DllImport attribute must be the DLL name.
    // The names of parameters are unimportant, but the types must be correct.
    [DllImport("myDLL.dll")]
    private static extern int add(int left, int right);

    static void Main(string[] args)
    {
        //The extern method can be called just as any other C# method.
        Console.WriteLine(add(1, 2));
    }
}
```

Vedi Calling convenzioni e manomissione dei nomi in C ++ per spiegazioni sul perché `extern "C"` e `__stdcall` siano necessari.

Trovare la libreria dinamica

Quando viene invocato per la prima volta il metodo extern, il programma C # cercherà e caricherà la DLL appropriata. Per ulteriori informazioni su dove viene eseguita la ricerca per trovare la DLL e su come è possibile influenzare le posizioni di ricerca, consultare questa domanda StackOverflow .

Codice semplice per esporre la classe per com

```
using System;
using System.Runtime.InteropServices;

namespace ComLibrary
{
    [ComVisible(true)]
    public interface IMainType
    {
        int GetInt();

        void StartTime();

        int StopTime();
    }

    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public class MainType : IMainType
    {
        private Stopwatch stopWatch;

        public int GetInt()
        {
            return 0;
        }

        public void StartTime()
        {
            stopWatch= new Stopwatch();
            stopWatch.Start();
        }

        public int StopTime()
        {
            return (int)stopWatch.ElapsedMilliseconds;
        }
    }
}
```

Mancanza di nomi in C ++

I compilatori C ++ codificano informazioni aggiuntive nei nomi delle funzioni esportate, come i tipi di argomenti, per rendere possibili sovraccarichi con argomenti diversi. Questo processo è chiamato mangling del nome . Ciò causa problemi con l'importazione di funzioni in C # (e l'interoperabilità con altre lingue in generale), poiché il nome della funzione `int add(int a, int b)` non viene più `add` , può essere `?add@@YAHHH@Z` , `_add@8` o qualsiasi altra cosa, a seconda del compilatore e della convenzione di chiamata.

Esistono diversi modi per risolvere il problema della manomissione dei nomi:

- Esportare funzioni usando `extern "C"` per passare al collegamento esterno C che utilizza il nome C mangling:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

  Il nome della funzione sarà ancora `_add@8` (`_add@8`), ma `StdCall` + `extern "C"` nome mangling è riconosciuto dal compilatore C #.

- Specifica dei nomi delle funzioni esportate nel file di definizione del modulo `myDLL.def` :

```
EXPORTS
  add
```

```
int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

  In questo caso, il nome della funzione sarà pura `add` .

- Importazione del nome mutilato. Avrai bisogno di un visualizzatore DLL per vedere il nome storpiato, quindi puoi specificarlo esplicitamente:

```
__declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll", EntryPoint = "?add@@YGHHH@Z")]
```

Chiamare convenzioni

Esistono diverse convenzioni delle funzioni di chiamata, che specificano chi (chiamante o chiamato) apre gli argomenti dallo stack, come vengono passati gli argomenti e in quale ordine. C ++ utilizza la convenzione di chiamata `Cdecl` per impostazione predefinita, ma C # si aspetta che `StdCall` venga utilizzato di solito dalle API di Windows. Devi cambiare uno o l'altro:

- Modifica convenzione di chiamata a `StdCall` in C ++:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

- In alternativa, modifica la convenzione di `Cdecl` in `Cdecl` in C #:

```
extern "C" __declspec(dllexport) int /*__cdecl*/ add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl)]
```

Se si desidera utilizzare una funzione con convenzione di chiamata `Cdecl` e un nome storpiato, il codice sarà simile a questo:

```
__declspec(dllexport) int add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl,
           EntryPoint = "?add@@YAHHH@Z")]
```

- **thiscall** (**__thiscall**) è usato principalmente in funzioni che sono membri di una classe.
- Quando una funzione utilizza **thiscall** (**__thiscall**), un puntatore alla classe viene passato come primo parametro.

Caricamento e scaricamento dinamico di DLL non gestite

Quando si utilizza l'attributo `DllImport` è necessario conoscere la dll corretta e il nome del metodo in *fase* di *compilazione* . Se si desidera essere più flessibili e decidere in *fase* di *esecuzione* quali DLL e metodi da caricare, è possibile utilizzare i metodi dell'API di Windows `LoadLibrary()` , `GetProcAddress()` e `FreeLibrary()` . Questo può essere utile se la libreria da utilizzare dipende dalle condizioni di runtime.

Il puntatore restituito da `GetProcAddress()` può essere convertito in un delegato utilizzando `Marshal.GetDelegateForFunctionPointer()` .

Il seguente esempio di codice lo dimostra con `myDLL.dll` degli esempi precedenti:

```
class Program
{
    // import necessary API as shown in other examples
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr LoadLibrary(string lib);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern void FreeLibrary(IntPtr module);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr module, string proc);

    // declare a delegate with the required signature
    private delegate int AddDelegate(int a, int b);

    private static void Main()
    {
        // load the dll
        IntPtr module = LoadLibrary("myDLL.dll");
        if (module == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load library: {Marshal.GetLastWin32Error()}");
            return;
        }

        // get a "pointer" to the method
        IntPtr method = GetProcAddress(module, "add");
```

```
        if (method == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load method: {Marshal.GetLastWin32Error()}");
            FreeLibrary(module);  // unload library
            return;
        }

        // convert "pointer" to delegate
        AddDelegate add = (AddDelegate)Marshal.GetDelegateForFunctionPointer(method,
typeof(AddDelegate));

        // use function
        int result = add(750, 300);

        // unload library
        FreeLibrary(module);
    }
}
```

Gestione degli errori Win32

Quando si utilizzano i metodi di interoperabilità, è possibile utilizzare l'API **GetLastError** per ottenere ulteriori informazioni sulle chiamate API dell'utente.

Attributo DllImport Attributo SetLastError

SetLastError = true

Indica che il callee chiamerà SetLastError (funzione API Win32).

SetLastError = false

Indica che il chiamato **non** chiamerà SetLastError (funzione API Win32), quindi non si otterranno informazioni di errore.

- Quando SetLastError non è impostato, è impostato su false (valore predefinito).
- È possibile ottenere il codice di errore utilizzando Marshal.GetLastWin32Error Method:

Esempio:

```
[DllImport("kernel32.dll", SetLastError=true)]
public static extern IntPtr OpenMutex(uint access, bool handle, string lpName);
```

Se si sta tentando di aprire il mutex che non esiste, GetLastError restituirà **ERROR_FILE_NOT_FOUND** .

```
var lastErrorCode = Marshal.GetLastWin32Error();

if (lastErrorCode == (uint)ERROR_FILE_NOT_FOUND)
{
    //Deal with error
}
```

I codici di errore del sistema possono essere trovati qui:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx

API GetLastError

Esiste un'API **GetLastError** nativa che puoi usare anche:

```
[DllImport("coredll.dll", SetLastError=true)]
static extern Int32 GetLastError();
```

- Quando si chiama l'API Win32 dal codice gestito, è necessario utilizzare sempre **Marshal.GetLastWin32Error** .

Ecco perché:

Tra la chiamata Win32 che imposta l'errore (chiama SetLastError), il CLR può chiamare anche altre chiamate Win32 che potrebbero chiamare **SetLastError** , questo comportamento può ignorare il valore dell'errore. In questo scenario, se si chiama **GetLastError** è possibile ottenere un errore non valido.

Impostazione **SetLastError = true** , assicura che il CLR recuperi il codice di errore prima di eseguire altre chiamate Win32.

Oggetto appuntato

GC (Garbage Collector) è responsabile della pulizia della nostra spazzatura.

Mentre **GC** pulisce la nostra spazzatura, rimuove gli oggetti non utilizzati dall'heap gestito che causano la frammentazione dell'heap. Quando **GC ha** terminato la rimozione, esegue una compressione heap (defragmintation) che implica lo spostamento di oggetti nell'heap.

Dato che **GC** non è deterministico, quando passa il riferimento / puntatore dell'oggetto gestito al codice nativo, **GC** può dare il via in qualsiasi momento, se si verifica subito dopo la chiamata Inerop, c'è una buona possibilità che l'oggetto (il riferimento passato a nativo) possa essere spostati sull'heap gestito - di conseguenza, otteniamo un riferimento non valido sul lato gestito.

In questo scenario, è necessario **bloccare** l'oggetto prima di passarlo al codice nativo.

Oggetto appuntato

L'oggetto appuntato è un oggetto che non è autorizzato a spostare da GC.

Maniglia appuntata Gc

Puoi creare un oggetto pin usando il metodo **Gc.Alloc**

```
GCHandle handle = GCHandle.Alloc(yourObject, GCHandleType.Pinned);
```

- Ottenere un **GCHandle aggiunto** all'oggetto gestito contrassegna un oggetto specifico

come uno che non può essere spostato da **GC** , fino a liberare l'handle

Esempio:

```
[DllImport("kernel32.dll", SetLastError = true)]
public static extern void EnterCriticalSection(IntPtr ptr);

[DllImport("kernel32.dll", SetLastError = true)]
public static extern void LeaveCriticalSection(IntPtr ptr);

public void EnterCriticalSection(CRITICAL_SECTION section)
{
    try
    {
        GCHandle handle = GCHandle.Alloc(section, GCHandleType.Pinned);
        EnterCriticalSection(handle.AddrOfPinnedObject());
        //Do Some Critical Work
        LeaveCriticalSection(handle.AddrOfPinnedObject());
    }
    finaly
    {
        handle.Free()
    }
}
```

Precauzioni

- Quando si immobilizza (specialmente oggetti di grandi dimensioni), provare a rilasciare il **GcHandle** bloccato il più velocemente possibile, poiché interrompe la deframmentazione dell'heap.
- Se dimentichi di liberare **GcHandle,** niente lo farà. Fallo in una sezione di codice sicura (come ad esempio finaly)

Leggere strutture con Maresciallo

La classe Marshal contiene una funzione denominata **PtrToStructure** , questa funzione ci dà la possibilità di leggere le strutture con un puntatore non gestito.

La funzione **PtrToStructure** ha molti sovraccarichi, ma tutti hanno la stessa intenzione.

PtrToStructure generico:

```
public static T PtrToStructure<T>(IntPtr ptr);
```

T - tipo di struttura.

ptr - Un puntatore a un blocco di memoria non gestito.

Esempio:

```
NATIVE_STRUCT result = Marshal.PtrToStructure<NATIVE_STRUCT>(ptr);
```

- Se hai a che fare con oggetti gestiti durante la lettura di strutture native, non dimenticare di

appuntare il tuo oggetto :)

```
T Read<T>(byte[] buffer)
   {
       T result = default(T);

       var gch = GCHandle.Alloc(buffer, GCHandleType.Pinned);

       try
       {
           result = Marshal.PtrToStructure<T>(gch.AddrOfPinnedObject());
       }
       finally
       {
           gch.Free();
       }

       return result;
   }
```

Leggi interoperabilità online: https://riptutorial.com/it/csharp/topic/3278/interoperabilita

Capitolo 87: Interpolazione a stringa

Sintassi

- $ "contenuto {espressione} contenuto"
- $ "contenuto {espressione: formato} contenuto"
- $ "contenuto {espressione} {{contenuto in parentesi}} contenuto}"
- $ "contenuto {espressione: formato} {{contenuto in parentesi}} contenuto}"

Osservazioni

L'interpolazione delle stringhe è una scorciatoia per il metodo `string.Format()` che semplifica la creazione di stringhe con valori di variabili ed espressioni al loro interno.

```
var name = "World";
var oldWay = string.Format("Hello, {0}!", name);  // returns "Hello, World"
var newWay = $"Hello, {name}!";                   // returns "Hello, World"
```

Examples

espressioni

Le espressioni complete possono anche essere utilizzate in stringhe interpolate.

```
var StrWithMathExpression = $"1 + 2 = {1 + 2}"; // -> "1 + 2 = 3"

string world = "world";
var StrWithFunctionCall = $"Hello, {world.ToUpper()}!"; // -> "Hello, WORLD!"
```

Live Demo su .NET Fiddle

Formatta le date nelle stringhe

```
var date = new DateTime(2015, 11, 11);
var str = $"It's {date:MMMM d, yyyy}, make a wish!";
System.Console.WriteLine(str);
```

È inoltre possibile utilizzare il metodo `DateTime.ToString` per formattare l'oggetto `DateTime` . Questo produrrà lo stesso risultato del codice sopra.

```
var date = new DateTime(2015, 11, 11);
var str = date.ToString("MMMM d, yyyy");
str = "It's " + str + ", make a wish!";
Console.WriteLine(str);
```

Produzione:

È l'11 novembre 2015, esprimi un desiderio!

Live Demo su .NET Fiddle

Demo live con DateTime.ToString

> **Nota:** `MM` indica mesi e `mm` per minuti. Fai molta attenzione quando li usi, perché gli errori possono introdurre bug che potrebbero essere difficili da scoprire.

Uso semplice

```
var name = "World";
var str = $"Hello, {name}!";
//str now contains: "Hello, World!";
```

Dietro le quinte

Internamente questo

```
$"Hello, {name}!"
```

Sarà compilato per qualcosa di simile a questo:

```
string.Format("Hello, {0}!", name);
```

Riempimento dell'output

La stringa può essere formattata in modo da accettare un parametro di riempimento che specificherà il numero di posizioni dei caratteri utilizzate dalla stringa inserita:

```
${value, padding}
```

> **NOTA: i** valori di riempimento positivo indicano che il riempimento sinistro e i valori di riempimento negativo indicano il riempimento destro.

Padding sinistro

Una spaziatura a sinistra di 5 (aggiunge 3 spazi prima del valore del numero, quindi occupa un totale di 5 posizioni di carattere nella stringa risultante).

```
var number = 42;
var str = $"The answer to life, the universe and everything is {number, 5}.";
//str is "The answer to life, the universe and everything is    42.";
//                                                          ^^^^^
System.Console.WriteLine(str);
```

Produzione:

```
The answer to life, the universe and everything is    42.
```

Live Demo su .NET Fiddle

Imbottitura a destra

Il riempimento destro, che utilizza un valore padding negativo, aggiungerà spazi alla fine del valore corrente.

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, -5}.";
//str is "The answer to life, the universe and everything is 42    .";
//                                                          ^^^^^
System.Console.WriteLine(str);
```

Produzione:

```
The answer to life, the universe and everything is 42    .
```

Live Demo su .NET Fiddle

Riempimento con specificatori di formato

Puoi anche usare gli specificatori di formattazione esistenti insieme al padding.

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, 5:f1}";
//str is "The answer to life, the universe and everything is 42.1 ";
//                                                          ^^^^^
```

Live Demo su .NET Fiddle

Formattare i numeri nelle stringhe

È possibile utilizzare i due punti e la sintassi del formato numerico standard per controllare la formattazione dei numeri.

```
var decimalValue = 120.5;

var asCurrency = $"It costs {decimalValue:C}";
// String value is "It costs $120.50" (depending on your local currency settings)

var withThreeDecimalPlaces = $"Exactly {decimalValue:F3}";
// String value is "Exactly 120.500"

var integerValue = 57;

var prefixedIfNecessary = $"{integerValue:D5}";
// String value is "00057"
```

Live Demo su .NET Fiddle

Leggi Interpolazione a stringa online: https://riptutorial.com/it/csharp/topic/22/interpolazione-a-stringa

Capitolo 88: iteratori

Osservazioni

Un iteratore è un metodo, ottiene accessor o operatore che esegue un'iterazione personalizzata su una matrice o una classe di raccolta utilizzando la parola chiave yield

Examples

Esempio di Iterator numerico semplice

Un caso d'uso comune per gli iteratori consiste nell'eseguire alcune operazioni su una serie di numeri. L'esempio seguente mostra come ogni elemento all'interno di una matrice di numeri può essere stampato individualmente sulla console.

Ciò è possibile perché gli array implementano l'interfaccia `IEnumerable` , consentendo ai client di ottenere un iteratore per l'array utilizzando il metodo `GetEnumerator()` . Questo metodo restituisce un *enumeratore* , che è un cursore di sola lettura, forward-only su ciascun numero nell'array.

```
int[] numbers = { 1, 2, 3, 4, 5 };

IEnumerator iterator = numbers.GetEnumerator();

while (iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
```

Produzione

```
1
2
3
4
5
```

È anche possibile ottenere gli stessi risultati usando una dichiarazione `foreach` :

```
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

Creazione di iteratori con rendimento

Iteratori *producono gli* enumeratori. In C #, gli enumeratori vengono prodotti definendo metodi, proprietà o indicizzatori che contengono dichiarazioni di `yield` .

La maggior parte dei metodi restituirà il controllo al chiamante tramite normali dichiarazioni di `return` , che dispone di tutto lo stato locale per tale metodo. Al contrario, i metodi che utilizzano le dichiarazioni di `yield` consentono loro di restituire più valori al chiamante su richiesta, *preservando allo stesso* tempo lo stato locale nel restituire tali valori. Questi valori restituiti costituiscono una sequenza. Esistono due tipi di dichiarazioni di `yield` utilizzate all'interno degli iteratori:

- `yield return` , che restituisce il controllo al chiamante ma mantiene lo stato. Il callee continuerà l'esecuzione da questa linea quando il controllo viene passato di nuovo ad esso.
- `yield break` , che funziona in modo simile a una normale dichiarazione di `return` - questo indica la fine della sequenza. Le dichiarazioni di `return` normali sono illegali all'interno di un blocco iteratore.

Questo esempio di seguito mostra un metodo iteratore che può essere utilizzato per generare la sequenza di Fibonacci :

```
IEnumerable<int> Fibonacci(int count)
{
    int prev = 1;
    int curr = 1;

    for (int i = 0; i < count; i++)
    {
        yield return prev;
        int temp = prev + curr;
        prev = curr;
        curr = temp;
    }
}
```

Questo iteratore può quindi essere utilizzato per produrre un enumeratore della sequenza di Fibonacci che può essere utilizzata da un metodo di chiamata. Il seguente codice mostra come possono essere enumerati i primi dieci termini all'interno della sequenza di Fibonacci:

```
void Main()
{
    foreach (int term in Fibonacci(10))
    {
        Console.WriteLine(term);
    }
}
```

Produzione

```
1
1
2
3
5
8
13
21
34
55
```

Leggi iteratori online: https://riptutorial.com/it/csharp/topic/4243/iteratori

Capitolo 89: La gestione delle eccezioni

Examples

Gestione delle eccezioni di base

```
try
{
    /* code that could throw an exception */
}
catch (Exception ex)
{
    /* handle the exception */
}
```

Si noti che la gestione di tutte le eccezioni con lo stesso codice spesso non è l'approccio migliore. Questo è comunemente usato quando qualsiasi routine di gestione delle eccezioni interne fallisce, come ultima risorsa.

Gestione di tipi di eccezione specifici

```
try
{
    /* code to open a file */
}
catch (System.IO.FileNotFoundException)
{
    /* code to handle the file being not found */
}
catch (System.IO.UnauthorizedAccessException)
{
    /* code to handle not being allowed access to the file */
}
catch (System.IO.IOException)
{
    /* code to handle IOException or it's descendant other than the previous two */
}
catch (System.Exception)
{
    /* code to handle other errors */
}
```

Fai attenzione che le eccezioni siano valutate in ordine e l'ereditarietà sia applicata. Quindi è necessario iniziare con quelli più specifici e terminare con il loro antenato. In qualsiasi punto, verrà eseguito un solo blocco catch.

Utilizzando l'oggetto eccezione

È consentito creare e generare eccezioni nel proprio codice. L'istanziazione di un'eccezione viene eseguita allo stesso modo di qualsiasi altro oggetto C #.

```
Exception ex = new Exception();

// constructor with an overload that takes a message string
Exception ex = new Exception("Error message");
```

È quindi possibile utilizzare la parola chiave `throw` per aumentare l'eccezione:

```
try
{
    throw new Exception("Error");
}
catch (Exception ex)
{
    Console.Write(ex.Message); // Logs 'Error' to the output window
}
```

Nota: se stai lanciando una nuova eccezione all'interno di un blocco catch, assicurati che l'eccezione originale sia passata come "eccezione interna", ad es

```
void DoSomething()
{
    int b=1; int c=5;
    try
    {
        var a = 1;
        b = a - 1;
        c = a / b;
        a = a / c;
    }
    catch (DivideByZeroException dEx) when (b==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by b because it is zero", dEx);
    }
    catch (DivideByZeroException dEx) when (c==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by c because it is zero", dEx);
    }
}

void Main()
{
    try
    {
        DoSomething();
    }
    catch (Exception ex)
    {
        // Logs full error information (incl. inner exception)
        Console.Write(ex.ToString());
    }
}
```

In questo caso si presume che l'eccezione non possa essere gestita, ma alcune informazioni utili vengono aggiunte al messaggio (e l'eccezione originale può ancora essere letta tramite `ex.InnerException` da un blocco di eccezioni esterno).

Mostrerà qualcosa come:

> System.DivideByZeroException: Non può dividere per b perché è zero --->
> System.DivideByZeroException: Tentativo di divisione per zero.
> a UserQuery.g__DoSomething0_0 () in C: [...] \ LINQPadQuery.cs: riga 36
> --- Fine della traccia dello stack delle eccezioni interne ---
> a UserQuery.g__DoSomething0_0 () in C: [...] \ LINQPadQuery.cs: riga 42
> a UserQuery.Main () in C: [...] \ LINQPadQuery.cs: riga 55

Se stai provando questo esempio in LinqPad, noterai che i numeri di riga non sono molto significativi (non sempre ti aiutano). Tuttavia, il passaggio di un testo di errore utile come suggerito sopra spesso riduce significativamente il tempo necessario per rintracciare la posizione dell'errore, che in questo esempio è chiaramente la linea

> c = a / b;

in funzione `DoSomething()` .

Provalo su .NET Fiddle

Finalmente blocco

```
try
{
    /* code that could throw an exception */
}
catch (Exception)
{
    /* handle the exception */
}
finally
{
    /* Code that will be executed, regardless if an exception was thrown / caught or not */
}
```

Il blocco `try / catch / finally` può essere molto utile durante la lettura da file.
Per esempio:

```
FileStream f = null;

try
{
    f = File.OpenRead("file.txt");
    /* process the file here */
}
finally
{
    f?.Close(); // f may be null, so use the null conditional operator.
}
```

Un blocco try deve essere seguito da un `catch` o da un blocco `finally` . Tuttavia, poiché non esiste un blocco catch, l'esecuzione causerà la chiusura. Prima della conclusione, verranno eseguite le istruzioni all'interno del blocco finally.

Nella lettura dei file avremmo potuto usare un blocco `using` come `FileStream` (ciò che `OpenRead` restituisce) implementa `IDisposable` .

Anche se c'è una dichiarazione di `return` nel blocco `try` , il blocco `finally` verrà eseguito di solito; ci sono alcuni casi in cui non lo faranno:

- Quando si verifica StackOverflow .
- `Environment.FailFast`
- Il processo di applicazione viene ucciso, di solito da una fonte esterna.

Implementazione di IErrorHandler per i servizi WCF

Implementare IErrorHandler per i servizi WCF è un ottimo modo per centralizzare la gestione degli errori e la registrazione. L'implementazione mostrata qui dovrebbe rilevare eventuali eccezioni non gestite generate a seguito di una chiamata a uno dei servizi WCF. Inoltre, in questo esempio viene mostrato come restituire un oggetto personalizzato e come restituire JSON piuttosto che l'XML predefinito.

Implementare IErrorHandler:

```
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Runtime.Serialization.Json;
using System.ServiceModel;
using System.ServiceModel.Web;

namespace BehaviorsAndInspectors
{
    public class ErrorHandler : IErrorHandler
    {

        public bool HandleError(Exception ex)
        {
            // Log exceptions here

            return true;

        } // end

        public void ProvideFault(Exception ex, MessageVersion version, ref Message fault)
        {
            // Get the outgoing response portion of the current context
            var response = WebOperationContext.Current.OutgoingResponse;

            // Set the default http status code
            response.StatusCode = HttpStatusCode.InternalServerError;

            // Add ContentType header that specifies we are using JSON
            response.ContentType = new MediaTypeHeaderValue("application/json").ToString();

            // Create the fault message that is returned (note the ref parameter) with
BaseDataResponseContract
            fault = Message.CreateMessage(
                version,
                string.Empty,
                new CustomReturnType { ErrorMessage = "An unhandled exception occurred!" },
```

```
                new DataContractJsonSerializer(typeof(BaseDataResponseContract), new
List<Type> { typeof(BaseDataResponseContract) }));

            if (ex.GetType() == typeof(VariousExceptionTypes))
            {
                 // You might want to catch different types of exceptions here and process
them differently
            }

            // Tell WCF to use JSON encoding rather than default XML
            var webBodyFormatMessageProperty = new
WebBodyFormatMessageProperty(WebContentFormat.Json);
            fault.Properties.Add(WebBodyFormatMessageProperty.Name,
webBodyFormatMessageProperty);

        } // end

    } // end class

} // end namespace
```

In questo esempio colleghiamo il gestore al comportamento del servizio. È inoltre possibile allegare ciò a IEndpointBehavior, IContractBehavior o IOperationBehavior in modo simile.

Allegare ai comportamenti di servizio:

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace BehaviorsAndInspectors
{
    public class ErrorHandlerExtension : BehaviorExtensionElement, IServiceBehavior
    {
        public override Type BehaviorType
        {
            get { return GetType(); }
        }

        protected override object CreateBehavior()
        {
            return this;
        }

        private IErrorHandler GetInstance()
        {
            return new ErrorHandler();
        }

        void IServiceBehavior.AddBindingParameters(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
BindingParameterCollection bindingParameters) { } // end

        void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase)
```

```
        {
            var errorHandlerInstance = GetInstance();

            foreach (ChannelDispatcher dispatcher in serviceHostBase.ChannelDispatchers)
            {
                dispatcher.ErrorHandlers.Add(errorHandlerInstance);
            }
        }

        void IServiceBehavior.Validate(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase) { } // end

    } // end class

} // end namespace
```

Config in Web.config:

```
...
<system.serviceModel>

    <services>
      <service name="WebServices.MyService">
        <endpoint binding="webHttpBinding" contract="WebServices.IMyService" />
      </service>
    </services>

    <extensions>
      <behaviorExtensions>
        <!-- This extension if for the WCF Error Handling-->
        <add name="ErrorHandlerBehavior"
type="WebServices.BehaviorsAndInspectors.ErrorHandlerExtensionBehavior, WebServices,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>

    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="true"/>
          <serviceDebug includeExceptionDetailInFaults="true"/>
          <ErrorHandlerBehavior />
        </behavior>
      </serviceBehaviors>
    </behaviors>

    ....
</system.serviceModel>
...
```

Ecco alcuni link che potrebbero essere utili su questo argomento:

https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler(v=vs.100).aspx

http://www.brainthud.com/cards/5218/25441/which-four-behavior-interfaces-exist-for-interacting-with-a-service-or-client-description-what-methods-do-they- attuare e

Altri esempi:

IErrorHandler che restituisce un corpo di messaggio errato quando il codice di stato HTTP è 401 Non autorizzato

IErrorHandler non sembra gestire i miei errori in WCF ... qualche idea?

Come rendere il gestore degli errori WCF personalizzato restituire la risposta JSON con codice http non OK?

Come si imposta l'intestazione Content-Type per una richiesta HttpClient?

Creazione di eccezioni personalizzate

È consentito implementare eccezioni personalizzate che possono essere generate come qualsiasi altra eccezione. Questo ha senso quando vuoi rendere le tue eccezioni distinguibili da altri errori durante il runtime.

In questo esempio creeremo un'eccezione personalizzata per la gestione chiara dei problemi che l'applicazione potrebbe avere durante l'analisi di un input complesso.

Creazione di classi di eccezioni personalizzate

Per creare un'eccezione personalizzata, crea una sottoclasse di `Exception` :

```
public class ParserException : Exception
{
    public ParserException() :
      base("The parsing went wrong and we have no additional information.") { }
}
```

L'eccezione personalizzata diventa molto utile quando si desidera fornire informazioni aggiuntive al catcher:

```
public class ParserException : Exception
{
    public ParserException(string fileName, int lineNumber) :
      base($"Parser error in {fileName}:{lineNumber}")
    {
      FileName = fileName;
      LineNumber = lineNumber;
    }
    public string FileName {get; private set;}
    public int LineNumber {get; private set;}
}
```

Ora, quando si `catch(ParserException x)` si avrà semantica aggiuntiva per ottimizzare la gestione delle eccezioni.

Le classi personalizzate possono implementare le seguenti funzionalità per supportare scenari aggiuntivi.

ri-lancio

Durante il processo di analisi, l'eccezione originale è ancora interessante. In questo esempio è un `FormatException` perché il codice tenta di analizzare un pezzo di stringa, che dovrebbe essere un numero. In questo caso l'eccezione personalizzata dovrebbe supportare l'inclusione di " **InnerException** ":

```
//new constructor:
ParserException(string msg, Exception inner) : base(msg, inner) {
}
```

serializzazione

In alcuni casi le eccezioni potrebbero dover attraversare i confini di AppDomain. Questo è il caso se il parser è in esecuzione nel proprio AppDomain per supportare il ricaricamento delle nuove configurazioni del parser. In Visual Studio, è possibile utilizzare il modello di `Exception` per generare codice come questo.

```
[Serializable]
public class ParserException : Exception
{
    // Constructor without arguments allows throwing your exception without
    // providing any information, including error message. Should be included
    // if your exception is meaningful without any additional details. Should
    // set message by calling base constructor (default message is not helpful).
    public ParserException()
        : base("Parser failure.")
    {}

    // Constructor with message argument allows overriding default error message.
    // Should be included if users can provide more helpful messages than
    // generic automatically generated messages.
    public ParserException(string message)
        : base(message)
    {}

    // Constructor for serialization support. If your exception contains custom
    // properties, read their values here.
    protected ParserException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {}
}
```

Utilizzo di ParserException

```
try
{
    Process.StartRun(fileName)
}
catch (ParserException ex)
{
    Console.WriteLine($"{ex.Message} in ${ex.FileName}:${ex.LineNumber}");
}
catch (PostProcessException x)
{
    ...
}
```

È inoltre possibile utilizzare eccezioni personalizzate per l'acquisizione e il wrapping delle eccezioni. In questo modo molti errori diversi possono essere convertiti in un singolo tipo di errore che è più utile all'applicazione:

```
try
{
    int foo = int.Parse(token);
}
catch (FormatException ex)
{
    //Assuming you added this constructor
    throw new ParserException(
      $"Failed to read {token} as number.",
      FileName,
      LineNumber,
      ex);
}
```

Quando si gestiscono eccezioni aumentando le proprie eccezioni personalizzate, si dovrebbe in genere includere un riferimento all'eccezione originale nella proprietà `InnerException` , come mostrato sopra.

Problemi di sicurezza

Se esporre il motivo dell'eccezione potrebbe compromettere la sicurezza consentendo agli utenti di vedere il funzionamento interno dell'applicazione, può essere una cattiva idea avvolgere l'eccezione interna. Questo potrebbe essere applicato se stai creando una libreria di classi che verrà utilizzata da altri.

Ecco come puoi generare un'eccezione personalizzata senza avvolgere l'eccezione interna:

```
try
{
  // ...
}
catch (SomeStandardException ex)
{
  // ...
  throw new MyCustomException(someMessage);
}
```

Conclusione

Quando si genera un'eccezione personalizzata (con wrapping o con una nuova eccezione unwrapped), è necessario generare un'eccezione significativa per il chiamante. Ad esempio, un utente di una libreria di classi potrebbe non sapere molto su come quella libreria faccia il suo lavoro interno. Le eccezioni generate dalle dipendenze della libreria di classi non sono significative. Piuttosto, l'utente desidera un'eccezione pertinente al modo in cui la libreria di classi utilizza tali dipendenze in modo errato.

```
try
{
  // ...
}
catch (IOException ex)
{
  // ...
  throw new StorageServiceException(@"The Storage Service encountered a problem saving
your data. Please consult the inner exception for technical details.
If you are not able to resolve the problem, please call 555-555-1234 for technical
assistance.", ex);
}
```

Eccezione anti-modelli

Ingerire le eccezioni

Si dovrebbe sempre ripetere l'eccezione nel seguente modo:

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw;
}
```

Rilanciare un'eccezione come quella di seguito offusca l'eccezione originale e perderà la traccia dello stack originale. Non si dovrebbe mai farlo! La traccia dello stack prima del fermo e del rethrow andrà persa.

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw ex;
```

```
}
```

Baseball Exception Handling

Non si dovrebbero usare le eccezioni come sostituto dei normali costrutti di controllo del flusso come le istruzioni if-then e i cicli while. Questo anti-pattern è talvolta chiamato Baseball Exception Handling .

Ecco un esempio di anti-pattern:

```
try
{
    while (AccountManager.HasMoreAccounts())
    {
        account = AccountManager.GetNextAccount();
        if (account.Name == userName)
        {
            //We found it
            throw new AccountFoundException(account);
        }
    }
}
catch (AccountFoundException found)
{
    Console.Write("Here are your account details: " + found.Account.Details.ToString());
}
```

Ecco un modo migliore per farlo:

```
Account found = null;
while (AccountManager.HasMoreAccounts() && (found==null))
{
    account = AccountManager.GetNextAccount();
    if (account.Name == userName)
    {
        //We found it
        found = account;
    }
}
Console.Write("Here are your account details: " + found.Details.ToString());
```

cattura (eccezione)

Ci sono quasi no (alcuni dicono nessuno!) Ragioni per catturare il tipo di eccezione generico nel codice. Dovresti prendere solo i tipi di eccezioni che ti aspetti di accadere, perché altrimenti nascondi bug nel tuo codice.

```
try
{
     var f = File.Open(myfile);
     // do something
```

```
}
catch (Exception x)
{
     // Assume file not found
     Console.Write("Could not open file");
     // but maybe the error was a NullReferenceException because of a bug in the file handling
code?
}
```

Meglio fare:

```
try
{
     var f = File.Open(myfile);
     // do something which should normally not throw exceptions
}
catch (IOException)
{
     Console.Write("File not found");
}
// Unfortunatelly, this one does not derive from the above, so declare separatelly
catch (UnauthorizedAccessException)
{
     Console.Write("Insufficient rights");
}
```

Se si verificano altre eccezioni, facciamo intenzionalmente in modo che l'applicazione si blocchi, quindi entra direttamente nel debugger e possiamo risolvere il problema. Non dobbiamo spedire un programma in cui siano presenti altre eccezioni che non avvengano comunque, quindi non è un problema avere un crash.

Anche il seguente è un cattivo esempio, poiché utilizza eccezioni per aggirare un errore di programmazione. Non è quello per cui sono progettati.

```
public void DoSomething(String s)
{
     if (s == null)
         throw new ArgumentNullException(nameof(s));
     // Implementation goes here
}

try
{
     DoSomething(myString);
}
catch(ArgumentNullException x)
{
    // if this happens, we have a programming error and we should check
    // why myString was null in the first place.
}
```

Eccezioni aggregate / più eccezioni da un metodo

Chi dice che non puoi lanciare più eccezioni con un solo metodo. Se non sei abituato a giocare con AggregateExceptions potresti essere tentato di creare la tua struttura dati per rappresentare

molte cose che vanno storte. Ovviamente ci sono altre strutture dati che non sono un'eccezione, ma sono ideali come i risultati di una convalida. Anche se giochi con AggregateExceptions potresti essere dal lato ricevente e gestirli sempre senza rendertene conto che possono esserti utili.

È abbastanza plausibile che un metodo venga eseguito e anche se sarà un fallimento nel suo insieme, vorrai evidenziare più cose che sono andate storte nelle eccezioni generate. Ad esempio, questo comportamento può essere visto con il modo in cui i metodi paralleli funzionano come un'attività suddivisa in più thread e qualsiasi numero di essi potrebbe generare eccezioni e questo deve essere segnalato. Ecco un esempio sciocco di come potresti trarne beneficio:

```
    public void Run()
    {
        try
        {
            this.SillyMethod(1, 2);
        }
        catch (AggregateException ex)
        {
            Console.WriteLine(ex.Message);
            foreach (Exception innerException in ex.InnerExceptions)
            {
                Console.WriteLine(innerException.Message);
            }
        }
    }

    private void SillyMethod(int input1, int input2)
    {
        var exceptions = new List<Exception>();

        if (input1 == 1)
        {
            exceptions.Add(new ArgumentException("I do not like ones"));
        }
        if (input2 == 2)
        {
            exceptions.Add(new ArgumentException("I do not like twos"));
        }
        if (exceptions.Any())
        {
            throw new AggregateException("Funny stuff happended during execution",
exceptions);
        }
    }
```

Annidamento di eccezioni e prova a catturare blocchi.

Uno è in grado di annidare un'eccezione / `try` blocco `catch` nell'altro.

In questo modo è possibile gestire piccoli blocchi di codice che sono in grado di funzionare senza interrompere l'intero meccanismo.

```
try
{
//some code here
```

```
    try
    {
        //some thing which throws an exception. For Eg : divide by 0
    }
    catch (DivideByZeroException dzEx)
    {
        //handle here only this exception
        //throw from here will be passed on to the parent catch block
    }
    finally
    {
        //any thing to do after it is done.
    }
 //resume from here & proceed as normal;
}
catch(Exception e)
{
    //handle here
}
```

Nota: evitare di ingerire le eccezioni quando si lancia sul blocco catch genitore

Migliori pratiche

Cheatsheet

FARE	NON
Controllo del flusso con istruzioni di controllo	Controlla il flusso con le eccezioni
Tieni traccia dell'eccezione ignorata (assorbita) mediante la registrazione	Ignora eccezione
Ripeti l'eccezione usando il `throw`	Rilanciare l'eccezione - `throw new ArgumentNullException()` o `throw ex`
Lancia eccezioni predefinite di sistema	Lancia eccezioni personalizzate simili alle eccezioni di sistema predefinite
Getta eccezioni personalizzate / predefinite se è fondamentale per la logica dell'applicazione	Getta eccezioni personalizzate / predefinite per indicare un avviso nel flusso
Individua le eccezioni che desideri gestire	Cattura ogni eccezione

NON gestire la logica aziendale con eccezioni.

Il controllo del flusso NON dovrebbe essere fatto da eccezioni. Utilizzare invece istruzioni condizionali. Se è possibile eseguire un controllo con l'istruzione `if-else` chiaro, non utilizzare eccezioni perché riduce la leggibilità e le prestazioni.

Considera il seguente frammento di Mr. Bad Practices:

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    Console.WriteLine(myObject.ToString());
}
```

Quando l'esecuzione raggiunge `Console.WriteLine(myObject.ToString());` l'applicazione genererà una NullReferenceException. Mr. Bad Practices ha capito che `myObject` è nullo e ha modificato il suo snippet per catturare e gestire `NullReferenceException` :

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    try
    {
        Console.WriteLine(myObject.ToString());
    }
    catch(NullReferenceException ex)
    {
        // Hmmm, if I create a new instance of object and assign it to myObject:
        myObject = new object();
        // Nice, now I can continue to work with myObject
        DoSomethingElseWithMyObject();
    }
}
```

Poiché lo snippet precedente riguarda solo la logica dell'eccezione, cosa devo fare se `myObject` non è nullo a questo punto? Dove dovrei coprire questa parte della logica? Subito dopo `Console.WriteLine(myObject.ToString());` ? Che ne dici dopo il `try...catch` bloccare il blocco?

Che ne dici di Mr. Best Practices? Come avrebbe gestito questo?

```
// This is a snippet example for DO
object myObject;
void DoingSomethingWithMyObject()
{
    if(myObject == null)
        myObject = new object();

    // When execution reaches this point, we are sure that myObject is not null
    DoSomethingElseWithMyObject();
}
```

Mr. Best Practices ha ottenuto la stessa logica con meno codice e una logica chiara e comprensibile.

NON ripetere le eccezioni

Rilanciare le eccezioni è costoso. Ha un impatto negativo sulle prestazioni. Per il codice che fallisce regolarmente, è possibile utilizzare modelli di progettazione per ridurre al minimo i problemi di prestazioni. Questo argomento descrive due schemi di progettazione utili quando le

eccezioni possono influire in modo significativo sulle prestazioni.

NON assorbire le eccezioni senza registrazione

```
try
{
    //Some code that might throw an exception
}
catch(Exception ex)
{
    //empty catch block, bad practice
}
```

Non ingoiare mai le eccezioni. Ignorare le eccezioni salverà quel momento ma creerà un caos per la manutenibilità in seguito. Quando si registrano le eccezioni, è necessario registrare sempre l'istanza dell'eccezione in modo che venga registrata la traccia dello stack completa e non solo il messaggio di eccezione.

```
try
{
    //Some code that might throw an exception
}
catch(NullException ex)
{
    LogManager.Log(ex.ToString());
}
```

Non prendere le eccezioni che non puoi gestire

Molte risorse, come questa , ti esortano fortemente a considerare il motivo per cui stai rilevando un'eccezione nel luogo in cui la stai catturando. Dovresti solo prendere un'eccezione se puoi gestirla in quella posizione. Se si può fare qualcosa per mitigare il problema, come provare un algoritmo alternativo, connettersi a un database di backup, provare un altro nome file, attendere 30 secondi e riprovare, o notificare un amministratore, è possibile rilevare l'errore e farlo. Se non c'è nulla che puoi plausibilmente e ragionevolmente fare, semplicemente "lascia andare" e lascia che l'eccezione sia gestita ad un livello più alto. Se l'eccezione è sufficientemente catastrofica e non vi è alcuna opzione ragionevole se non che l'intero programma si arresti in modo anomalo a causa della gravità del problema, allora lascialo andare in crash.

```
try
{
    //Try to save the data to the main database.
}
catch(SqlException ex)
{
    //Try to save the data to the alternative database.
}
//If anything other than a SqlException is thrown, there is nothing we can do here. Let the
exception bubble up to a level where it can be handled.
```

Eccezione non gestita e thread

AppDomain.UnhandledException Questo evento fornisce la notifica delle eccezioni non rilevate. Consente all'applicazione di registrare le informazioni sull'eccezione prima che il gestore predefinito del sistema segnali l'eccezione all'utente e interrompa l'applicazione. Se sono disponibili informazioni sufficienti sullo stato dell'applicazione, altre le azioni possono essere intraprese, ad esempio il salvataggio dei dati del programma per il successivo recupero. Si consiglia di procedere, poiché i dati del programma possono essere danneggiati quando le eccezioni non vengono gestite.

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
    }
```

Application.ThreadException Questo evento consente all'applicazione Windows Form di gestire eccezioni altrimenti non gestite che si verificano nei thread di Windows Form. Allegare i gestori di eventi all'evento ThreadException per gestire queste eccezioni, che lasceranno l'applicazione in uno stato sconosciuto. Ove possibile, le eccezioni dovrebbero essere gestite da un blocco strutturato di gestione delle eccezioni.

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
        Application.ThreadException += new ThreadExceptionEventHandler(ThreadException);
    }
```

E infine la gestione delle eccezioni

```
static void UnhandledException(object sender, UnhandledExceptionEventArgs e)
    {
        Exception ex = (Exception)e.ExceptionObject;
        // your code
    }

static void ThreadException(object sender, ThreadExceptionEventArgs e)
    {
        Exception ex = e.Exception;
        // your code
    }
```

Lanciare un'eccezione

Il tuo codice può, e spesso dovrebbe, generare un'eccezione quando qualcosa di insolito è successo.

```
public void WalkInto(Destination destination)
{
    if (destination.Name == "Mordor")
    {
        throw new InvalidOperationException("One does not simply walk into Mordor.");
    }
    // ... Implement your normal walking code here.
}
```

Leggi La gestione delle eccezioni online: https://riptutorial.com/it/csharp/topic/40/la-gestione-delle-eccezioni

Capitolo 90: Lambda Expressions

Osservazioni

chiusure

Le espressioni lambda catturano implicitamente le variabili utilizzate e creano una chiusura . Una chiusura è una funzione insieme a un contesto statale. Il compilatore genererà una chiusura ogni volta che un'espressione lambda "racchiude" un valore dal suo contesto circostante.

Ad esempio quando viene eseguito quanto segue

```
Func<object, bool> safeApplyFiltererPredicate = o => (o != null) && filterer.Predicate(i);
```

`safeApplyFilterPredicate` fa riferimento a un oggetto appena creato che ha un riferimento privato al valore corrente del `filterer` e il cui metodo `Invoke` si comporta come

```
o => (o != null) && filterer.Predicate(i);
```

Questo può essere importante, poiché finché viene mantenuto il riferimento al valore ora in `safeApplyFilterPredicate` , ci sarà un riferimento all'oggetto cui si riferisce attualmente il `filterer` . Ciò ha un effetto sulla garbage collection e può causare comportamenti imprevisti se l'oggetto al quale il `filterer` riferisce attualmente è mutato.

D'altra parte, le chiusure possono essere utilizzate per deliberare l'effetto di incapsulare un comportamento che implica riferimenti ad altri oggetti.

Per esempio

```
var logger = new Logger();
Func<int, int> Add1AndLog = i => {
    logger.Log("adding 1 to " + i);
    return (i + 1);
};
```

Le chiusure possono anche essere utilizzate per modellare macchine a stati:

```
Func<int, int> MyAddingMachine() {
    var i = 0;
    return x => i += x;
};
```

Examples

Espressioni lambda di base

```
Func<int, int> add1 = i => i + 1;

Func<int, int, int> add = (i, j) => i + j;

// Behaviourally equivalent to:

int Add1(int i)
{
    return i + 1;
}

int Add(int i, int j)
{
    return i + j;
}

...

Console.WriteLine(add1(42)); //43
Console.WriteLine(Add1(42)); //43
Console.WriteLine(add(100, 250)); //350
Console.WriteLine(Add(100, 250)); //350
```

Espressioni lambda di base con LINQ

```
// assume source is {0, 1, 2, ..., 10}

var evens = source.Where(n => n%2 == 0);
// evens = {0, 2, 4, ... 10}

var strings = source.Select(n => n.ToString());
// strings = {"0", "1", ..., "10"}
```

Usare la sintassi lambda per creare una chiusura

Vedi le osservazioni per la discussione delle chiusure. Supponiamo di avere un'interfaccia:

```
public interface IMachine<TState, TInput>
{
    TState State { get; }
    public void Input(TInput input);
}
```

e quindi viene eseguito quanto segue:

```
IMachine<int, int> machine = ...;
Func<int, int> machineClosure = i => {
    machine.Input(i);
    return machine.State;
};
```

Ora `machineClosure` fa riferimento a una funzione da `int` a `int` , che dietro le quinte usa l'istanza `IMachine` cui fa riferimento la `machine` per eseguire il calcolo. Anche se la `machine` riferimento esce dall'ambito, finché l'oggetto `machineClosure` viene mantenuto, l'istanza `IMachine` originale verrà

mantenuta come parte di una "chiusura", definita automaticamente dal compilatore.

Attenzione: questo può significare che la stessa chiamata di funzione restituisce valori diversi in momenti diversi (es. In questo esempio se la macchina mantiene una somma dei suoi ingressi). In molti casi, questo può essere inaspettato e deve essere evitato per qualsiasi codice in uno stile funzionale: le chiusure accidentali e inaspettate possono essere una fonte di bug.

Sintassi Lambda con corpo del blocco di istruzioni

```
Func<int, string> doubleThenAddElevenThenQuote = i => {
    var doubled = 2 * i;
    var addedEleven = 11 + doubled;
    return $"'{addedEleven}'";
};
```

Espressioni Lambda con System.Linq.Expressions

```
Expression<Func<int, bool>> checkEvenExpression = i => i%2 == 0;
// lambda expression is automatically converted to an Expression<Func<int, bool>>
```

Leggi Lambda Expressions online: https://riptutorial.com/it/csharp/topic/7057/lambda-expressions

Capitolo 91: Le tuple

Examples

Creare tuple

Le tuple sono create usando tipi generici `Tuple<T1>` - `Tuple<T1,T2,T3,T4,T5,T6,T7,T8>` . Ciascuno dei tipi rappresenta una tupla contenente da 1 a 8 elementi. Gli elementi possono essere di diversi tipi.

```
// tuple with 4 elements
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
```

È inoltre possibile creare `Tuple.Create` usando i metodi statici `Tuple.Create` . In questo caso, i tipi di elementi vengono dedotti dal compilatore C #.

```
// tuple with 4 elements
var tuple = Tuple.Create("foo", 123, true, new MyClass());
```

7.0

Dal momento che C # 7.0, le tuple possono essere facilmente create usando ValueTuple .

```
var tuple = ("foo", 123, true, new MyClass());
```

Gli elementi possono essere nominati per facilitare la decomposizione.

```
(int number, bool flag, MyClass instance) tuple = (123, true, new MyClass());
```

Accesso agli elementi di tuple

Per accedere agli elementi tuple utilizzano `Item1` - `Item8` proprietà. Saranno disponibili solo le proprietà con un numero di indice inferiore o uguale alla dimensione della tupla (ovvero non è possibile accedere `Item3` proprietà `Item3` in `Tuple<T1,T2>`).

```
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
var item1 = tuple.Item1; // "foo"
var item2 = tuple.Item2; // 123
var item3 = tuple.Item3; // true
var item4 = tuple.Item4; // new My Class()
```

Confronto e ordinamento delle tuple

Le tuple possono essere confrontate in base ai loro elementi.

Ad esempio, una enumerabile i cui elementi sono di tipo `Tuple` può essere ordinata in base agli

operatori di confronto definiti su un elemento specificato:

```
List<Tuple<int, string>> list = new List<Tuple<int, string>>();
list.Add(new Tuple<int, string>(2, "foo"));
list.Add(new Tuple<int, string>(1, "bar"));
list.Add(new Tuple<int, string>(3, "qux"));

list.Sort((a, b) => a.Item2.CompareTo(b.Item2)); //sort based on the string element

foreach (var element in list) {
    Console.WriteLine(element);
}

// Output:
// (1, bar)
// (2, foo)
// (3, qux)
```

O per invertire l'uso di tipo:

```
list.Sort((a, b) => b.Item2.CompareTo(a.Item2));
```

Restituisce più valori da un metodo

Le tuple possono essere utilizzate per restituire più valori da un metodo senza utilizzare parametri. Nell'esempio seguente, `AddMultiply` viene utilizzato per restituire due valori (somma, prodotto).

```
void Write()
{
    var result = AddMultiply(25, 28);
    Console.WriteLine(result.Item1);
    Console.WriteLine(result.Item2);
}

Tuple<int, int> AddMultiply(int a, int b)
{
    return new Tuple<int, int>(a + b, a * b);
}
```

Produzione:

53
700

Ora C # 7.0 offre un modo alternativo per restituire più valori dai metodi che utilizzano le tuple valore Ulteriori informazioni sulla struttura `ValueTuple` .

Leggi Le tuple online: https://riptutorial.com/it/csharp/topic/838/le-tuple

Capitolo 92: Leggi e capisci Stacktraces

introduzione

Una traccia dello stack è di grande aiuto durante il debug di un programma. Si otterrà una traccia stack quando il programma genera un'eccezione e talvolta quando il programma termina in modo anomalo.

Examples

Traccia dello stack per una semplice NullReferenceException in Windows Form

Creiamo un piccolo pezzo di codice che genera un'eccezione:

```
private void button1_Click(object sender, EventArgs e)
{
    string msg = null;
    msg.ToCharArray();
}
```

Se lo eseguiamo, otteniamo la seguente eccezione e traccia dello stack:

```
System.NullReferenceException: "Object reference not set to an instance of an object."
   at WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) in
F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:line 29
   at System.Windows.Forms.Control.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnMouseUp(MouseEventArgs mevent)
```

La traccia dello stack continua così, ma questa parte è sufficiente per i nostri scopi.

Nella parte superiore della traccia dello stack vediamo la linea:

> a WindowsFormsApplication1.Form1.button1_Click (Oggetto mittente, EventArgs e) in F: \ WindowsFormsApplication1 \ WindowsFormsApplication1 \ Form1.cs: riga 29

Questa è la parte più importante. Indica la riga *esatta in* cui si è verificata l'eccezione: riga 29 in Form1.cs.
Quindi, qui inizia la tua ricerca.

La seconda linea è

> a System.Windows.Forms.Control.OnClick (EventArgs e)

Questo è il metodo che ha chiamato `button1_Click` . Così ora sappiamo che `button1_Click` , dove si è verificato l'errore, è stato chiamato da `System.Windows.Forms.Control.OnClick` .

Possiamo continuare così; la terza linea è

> a System.Windows.Forms.Button.OnClick (EventArgs e)

Questo è, a sua volta, il codice che ha chiamato `System.windows.Forms.Control.OnClick` .

La traccia dello stack è l'elenco di funzioni che è stato chiamato fino a quando il codice non ha incontrato l'eccezione. E seguendo questo, puoi capire quale percorso di esecuzione ha seguito il tuo codice fino a quando non si è trovato nei guai!

Si noti che la traccia dello stack include chiamate dal sistema .Net; normalmente non è necessario seguire tutti i codici Microsofts `System.Windows.Forms` per scoprire cosa è andato storto, solo il codice che appartiene alla propria applicazione.

Quindi, perché si chiama "stack trace"?
Perché ogni volta che un programma chiama un metodo, tiene traccia di dove si trovava. Ha una struttura dati chiamata "stack", dove scarica la sua ultima posizione.
Se è fatto eseguendo il metodo, guarda nello stack per vedere dove era prima che chiamasse il metodo - e continua da lì.

Quindi lo stack consente al computer di sapere dove è stato interrotto prima di chiamare un nuovo metodo.

Ma serve anche come aiuto per il debug. Come un detective che traccia i passi compiuti da un criminale quando commette il crimine, un programmatore può usare lo stack per tracciare i passi compiuti da un programma prima che si schiantasse.

Leggi Leggi e capisci Stacktraces online: https://riptutorial.com/it/csharp/topic/8923/leggi-e-capisci-stacktraces

Capitolo 93: letterali

Sintassi

- **bool:** vero o falso
- **byte:** None, letterale intero convertito implicitamente da int
- **sbyte:** Nessuno, letterale intero convertito implicitamente da int
- **char:** avvolgere il valore con virgolette singole
- **decimale:** M o m
- **doppio:** D, d o un numero reale
- **float:** F or f
- **int:** None, predefinito per i valori interi all'interno dell'intervallo di int
- **uint:** U, u, o valori interi nell'intervallo di uint
- **long:** L, l, o valori interi nell'intervallo di long
- **ulong:** UL, ul, Ul, uL, LU, lu, Lu, lU, o valori interi all'interno della gamma di ulong
- **short:** None, letterale intero convertito implicitamente da int
- **ushort:** None, letterale intero convertito implicitamente da int
- **stringa:** avvolgi il valore con virgolette, opzionalmente aggiunto con `@`
- **null** : il valore letterale `null`

Examples

letterali int

`int` valori letterali `int` vengono definiti semplicemente utilizzando valori interi all'interno dell'intervallo di `int` :

```
int i = 5;
```

uint letterali

`uint` letterali sono definite utilizzando il suffisso `U` o `u` , oppure utilizzando un valore integrale nell'intervallo `uint` :

```
uint ui = 5U;
```

stringhe letterali

`string` letterali sono definite avvolgendo il valore con virgolette `"` :

```
string s = "hello, this is a string literal";
```

I valori letterali stringa possono contenere sequenze di escape. Vedi Sequenze di escape delle stringhe

Inoltre, C # supporta letterali stringa letterali (vedi Stringhe di Verbatim). Questi sono definiti avvolgendo il valore con virgolette `"` , e anteponendolo a `@` . Le sequenze di escape vengono ignorate in letterali stringa letterali e sono inclusi tutti i caratteri di spaziatura:

```
string s = @"The path is:
C:\Windows\System32";
//The backslashes and newline are included in the string
```

letterali di carbone

`char` letterali sono definite avvolgendo il valore con apici singoli `'` :

```
char c = 'h';
```

I caratteri letterali possono contenere sequenze di escape. Vedi Sequenze di escape delle stringhe

Un carattere letterale deve essere esattamente un carattere (dopo che tutte le sequenze di escape sono state valutate). I caratteri letterali vuoti non sono validi. Il carattere predefinito (restituito per `default(char)` o `new char()`) è `'\0'` , o il carattere NULL (da non confondere con i riferimenti `null` letterali e null).

letterali byte

`byte` tipo di `byte` non ha suffisso letterale. I valori letterali interi vengono convertiti implicitamente da `int` :

```
byte b = 127;
```

letterali sbyte

`sbyte` tipo `sbyte` non ha suffisso letterale. I valori letterali interi vengono convertiti implicitamente da `int` :

```
sbyte sb = 127;
```

letterali decimali

`decimal` valori letterali `decimal` sono definiti utilizzando il suffisso M o m su un numero reale:

```
decimal m = 30.5M;
```

doppi letterali

`double` valori letterali `double` sono definiti utilizzando il suffisso D o d oppure utilizzando un numero reale:

```
double d = 30.5D;
```

letterali galleggianti

`float` letterali `float` vengono definiti utilizzando il suffisso F o f o utilizzando un numero reale:

```
float f = 30.5F;
```

letterali lunghi

`long` letterali `long` sono definiti utilizzando il suffisso `L` o `l` , oppure utilizzando valori interi nell'intervallo di `long` :

```
long l = 5L;
```

molto letterale

`ulong` letterali `ulong` sono definiti usando il suffisso `UL` , `ul` , `Ul` , `uL` , `LU` , `lu` , `Lu` o `lU` , oppure usando valori interi all'interno dell'intervallo di `ulong` :

```
ulong ul = 5UL;
```

breve letterale

`short` tipo `short` non ha valore letterale. I valori letterali interi vengono convertiti implicitamente da `int` :

```
short s = 127;
```

Ushort letterale

tipo di `ushort` non ha suffisso letterale. I valori letterali interi vengono convertiti implicitamente da `int` :

```
ushort us = 127;
```

bool letterali

`bool` valori letterali `bool` sono `true` o `false` ;

```
bool b = true;
```

Leggi letterali online: https://riptutorial.com/it/csharp/topic/2655/letterali

Capitolo 94: Lettura e scrittura di file .zip

Sintassi

1. public static ZipArchive OpenRead (string archiveFileName)

Parametri

Parametro	Dettagli
archivefilename	Il percorso dell'archivio da aprire, specificato come percorso relativo o assoluto. Un percorso relativo viene interpretato come relativo alla directory di lavoro corrente.

Examples

Scrivere in un file zip

Per scrivere un nuovo file .zip:

```
System.IO.Compression
System.IO.Compression.FileSystem

using (FileStream zipToOpen = new FileStream(@"C:\temp", FileMode.Open))
{
    using (ZipArchive archive = new ZipArchive(zipToOpen, ZipArchiveMode.Update))
    {
        ZipArchiveEntry readmeEntry = archive.CreateEntry("Readme.txt");
        using (StreamWriter writer = new StreamWriter(readmeEntry.Open()))
        {
            writer.WriteLine("Information about this package.");
            writer.WriteLine("========================");
        }
    }
}
```

Scrittura di file zip in memoria

L'esempio seguente restituirà i dati `byte[]` di un file zippato contenente i file forniti, senza bisogno di accedere al file system.

```
public static byte[] ZipFiles(Dictionary<string, byte[]> files)
{
    using (MemoryStream ms = new MemoryStream())
    {
        using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Update))
        {
            foreach (var file in files)
```

```
            {
                ZipArchiveEntry orderEntry = archive.CreateEntry(file.Key); //create a file
with this name
                using (BinaryWriter writer = new BinaryWriter(orderEntry.Open()))
                {
                    writer.Write(file.Value); //write the binary data
                }
            }
        }
        //ZipArchive must be disposed before the MemoryStream has data
        return ms.ToArray();
    }
}
```

Ottieni file da un file zip

Questo esempio ottiene un elenco di file dai dati binari dell'archivio zip forniti:

```
public static Dictionary<string, byte[]> GetFiles(byte[] zippedFile)
{
    using (MemoryStream ms = new MemoryStream(zippedFile))
    using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Read))
    {
        return archive.Entries.ToDictionary(x => x.FullName, x => ReadStream(x.Open()));
    }
}

private static byte[] ReadStream(Stream stream)
{
    using (var ms = new MemoryStream())
    {
        stream.CopyTo(ms);
        return ms.ToArray();
    }
}
```

L'esempio seguente mostra come aprire un archivio zip ed estrarre tutti i file .txt in una cartella

```
using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string zipPath = @"c:\example\start.zip";
            string extractPath = @"c:\example\extract";

            using (ZipArchive archive = ZipFile.OpenRead(zipPath))
            {
                foreach (ZipArchiveEntry entry in archive.Entries)
                {
                    if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
```

```
                    {
                        entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
                    }
                }
            }
        }
    }
}
```

Leggi Lettura e scrittura di file .zip online: https://riptutorial.com/it/csharp/topic/6709/lettura-e-scrittura-di-file--zip

Capitolo 95: LINQ in XML

Examples

Leggi XML usando LINQ in XML

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
 <Employee>
  <EmpId>1</EmpId>
  <Name>Sam</Name>
  <Sex>Male</Sex>
  <Phone Type="Home">423-555-0124</Phone>
  <Phone Type="Work">424-555-0545</Phone>
  <Address>
   <Street>7A Cox Street</Street>
   <City>Acampo</City>
   <State>CA</State>
   <Zip>95220</Zip>
   <Country>USA</Country>
  </Address>
</Employee>
<Employee>
 <EmpId>2</EmpId>
 <Name>Lucy</Name>
 <Sex>Female</Sex>
 <Phone Type="Home">143-555-0763</Phone>
 <Phone Type="Work">434-555-0567</Phone>
 <Address>
  <Street>Jess Bay</Street>
  <City>Alta</City>
  <State>CA</State>
  <Zip>95701</Zip>
  <Country>USA</Country>
  </Address>
 </Employee>
</Employees>
```

Per leggere quel file XML usando LINQ

```
XDocument xdocument = XDocument.Load("Employees.xml");
IEnumerable<XElement> employees = xdocument.Root.Elements();
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```

Per accedere a un singolo elemento

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names :");
foreach (var employee in employees)
{
```

```
    Console.WriteLine(employee.Element("Name").Value);
}
```

Per accedere a più elementi

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names along with their ID:");
foreach (var employee in employees)
{
    Console.WriteLine("{0} has Employee ID {1}",
    employee.Element("Name").Value,
    employee.Element("EmpId").Value);
}
```

Per accedere a tutti gli elementi che hanno un attributo specifico

```
XElement xelement = XElement.Load("Employees.xml");
var name = from nm in xelement.Root.Elements("Employee")
       where (string)nm.Element("Sex") == "Female"
       select nm;
Console.WriteLine("Details of Female Employees:");
foreach (XElement xEle in name)
Console.WriteLine(xEle);
```

Per accedere a un elemento specifico che ha un attributo specifico

```
XElement xelement = XElement.Load("..\\..\\Employees.xml");
var homePhone = from phoneno in xelement.Root.Elements("Employee")
            where (string)phoneno.Element("Phone").Attribute("Type") == "Home"
            select phoneno;
Console.WriteLine("List HomePhone Nos.");
foreach (XElement xEle in homePhone)
{
    Console.WriteLine(xEle.Element("Phone").Value);
}
```

Leggi LINQ in XML online: https://riptutorial.com/it/csharp/topic/2773/linq-in-xml

Capitolo 96: LINQ parallelo (PLINQ)

Sintassi

- ParallelEnumerable.Aggregate (func)
- ParallelEnumerable.Aggregate (seed, func)
- ParallelEnumerable.Aggregate (seed, updateAccumulatorFunc, combineAccumulatorsFunc, resultSelector)
- ParallelEnumerable.Aggregate (seedFactory, updateAccumulatorFunc, combineAccumulatorsFunc, resultSelector)
- ParallelEnumerable.All (predicato)
- ParallelEnumerable.Any ()
- ParallelEnumerable.Any (predicato)
- ParallelEnumerable.AsEnumerable ()
- ParallelEnumerable.AsOrdered ()
- ParallelEnumerable.AsParallel ()
- ParallelEnumerable.AsSequential ()
- ParallelEnumerable.AsUnordered ()
- ParallelEnumerable.Average (selettore)
- ParallelEnumerable.Cast ()
- ParallelEnumerable.Concat (secondo)
- ParallelEnumerable.Contains (valore)
- ParallelEnumerable.Contains (valore, comparatore)
- ParallelEnumerable.Count ()
- ParallelEnumerable.Count (predicato)
- ParallelEnumerable.DefaultIfEmpty ()
- ParallelEnumerable.DefaultIfEmpty (defaultValue)
- ParallelEnumerable.Distinct ()
- ParallelEnumerable.Distinct (di confronto)
- ParallelEnumerable.ElementAt (indice)
- ParallelEnumerable.ElementAtOrDefault (indice)
- ParallelEnumerable.Empty ()
- ParallelEnumerable.Except (secondo)
- ParallelEnumerable.Except (second, comparer)
- ParallelEnumerable.First ()
- ParallelEnumerable.First (predicato)
- ParallelEnumerable.FirstOrDefault ()
- ParallelEnumerable.FirstOrDefault (predicato)
- ParallelEnumerable.ForAll (azione)
- ParallelEnumerable.GroupBy (keySelector)
- ParallelEnumerable.GroupBy (keySelector, comparatore)
- ParallelEnumerable.GroupBy (keySelector, elementSelector)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, comparatore)
- ParallelEnumerable.GroupBy (keySelector, resultSelector)

- ParallelEnumerable.GroupBy (keySelector, resultSelector, comparatore)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, ruleSelector)
- ParallelEnumerable.GroupBy (keySelector, elementSelector, ruleSelector, comparatore)
- ParallelEnumerable.GroupJoin (inner, outerKeySelector, innerKeySelector, resultSelector)
- ParallelEnumerable.GroupJoin (inner, outerKeySelector, innerKeySelector, resultSelector, comparatore)
- ParallelEnumerable.Intersect (secondo)
- ParallelEnumerable.Intersect (secondo, comparatore)
- ParallelEnumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector)
- ParallelEnumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector, comparatore)
- ParallelEnumerable.Last ()
- ParallelEnumerable.Last (predicato)
- ParallelEnumerable.LastOrDefault ()
- ParallelEnumerable.LastOrDefault (predicato)
- ParallelEnumerable.LongCount ()
- ParallelEnumerable.LongCount (predicato)
- ParallelEnumerable.Max ()
- ParallelEnumerable.Max (selettore)
- ParallelEnumerable.Min ()
- ParallelEnumerable.Min (selettore)
- ParallelEnumerable.OfType ()
- ParallelEnumerable.OrderBy (keySelector)
- ParallelEnumerable.OrderBy (keySelector, comparatore)
- ParallelEnumerable.OrderByDescending (keySelector)
- ParallelEnumerable.OrderByDescending (keySelector, comparatore)
- ParallelEnumerable.Range (start, count)
- ParallelEnumerable.Repeat (element, count)
- ParallelEnumerable.Reverse ()
- ParallelEnumerable.Select (selettore)
- ParallelEnumerable.SelectMany (selettore)
- ParallelEnumerable.SelectMany (collectionSelector, resultSelector)
- ParallelEnumerable.SequenceEqual (secondo)
- ParallelEnumerable.SequenceEqual (secondo, comparatore)
- ParallelEnumerable.Single ()
- ParallelEnumerable.Single (predicato)
- ParallelEnumerable.SingleOrDefault ()
- ParallelEnumerable.SingleOrDefault (predicato)
- ParallelEnumerable.Skip (conteggio)
- ParallelEnumerable.SkipWhile (predicato)
- ParallelEnumerable.Sum ()
- ParallelEnumerable.Sum (selettore)
- ParallelEnumerable.Take (conteggio)
- ParallelEnumerable.TakeWhile (predicato)
- ParallelEnumerable.ThenBy (keySelector)
- ParallelEnumerable.ThenBy (keySelector, comparatore)

- ParallelEnumerable.ThenByDescending (keySelector)
- ParallelEnumerable.ThenByDescending (keySelector, comparatore)
- ParallelEnumerable.ToArray ()
- ParallelEnumerable.ToDictionary (keySelector)
- ParallelEnumerable.ToDictionary (keySelector, comparatore)
- ParallelEnumerable.ToDictionary (elementSelector)
- ParallelEnumerable.ToDictionary (elementSelector, comparatore)
- ParallelEnumerable.ToList ()
- ParallelEnumerable.ToLookup (keySelector)
- ParallelEnumerable.ToLookup (keySelector, comparatore)
- ParallelEnumerable.ToLookup (keySelector, elementSelector)
- ParallelEnumerable.ToLookup (keySelector, elementSelector, comparatore)
- ParallelEnumerable.Union (secondo)
- ParallelEnumerable.Union (secondo, comparatore)
- ParallelEnumerable.Where (predicato)
- ParallelEnumerable.WithCancellation (CancellationToken)
- ParallelEnumerable.WithDegreeOfParallelism (degreeOfParallelism)
- ParallelEnumerable.WithExecutionMode (executionMode)
- ParallelEnumerable.WithMergeOptions (mergeOptions)
- ParallelEnumerable.Zip (second, resultSelector)

Examples

Semplice esempio

Questo esempio mostra come PLINQ può essere utilizzato per calcolare i numeri pari tra 1 e 10.000 utilizzando più thread. Si noti che la lista risultante non sarà ordinata!

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 4, 26, 28, 30, ... }
// Order will vary with different runs
```

WithDegreeOfParallelism

Il grado di parallelismo è il numero massimo di attività di esecuzione simultanee che verranno utilizzate per elaborare la query.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .WithDegreeOfParallelism(4)
                          .Where(x => x % 2 == 0);
```

AsOrdered

Questo esempio mostra come PLINQ può essere utilizzato per calcolare i numeri pari tra 1 e 10.000 utilizzando più thread. L'ordine verrà mantenuto nell'elenco risultante, tuttavia tieni presente che `AsOrdered` può danneggiare le prestazioni per un numero elevato di elementi, pertanto l'elaborazione non ordinata viene preferita quando possibile.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .AsOrdered()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 2, 4, 6, 8, ..., 10000 }
```

AsUnordered

Le sequenze ordinate possono danneggiare le prestazioni quando si ha a che fare con un gran numero di elementi. Per mitigarlo, è possibile chiamare `AsUnordered` quando l'ordine della sequenza non è più necessario.

```
var sequence = Enumerable.Range(1, 10000).Select(x => -1 * x); // -1, -2, ...
var evenNumbers = sequence.AsParallel()
                          .OrderBy(x => x)
                          .Take(5000)
                          .AsUnordered()
                          .Where(x => x % 2 == 0) // This line won't be affected by ordering
                          .ToList();
```

Leggi LINQ parallelo (PLINQ) online: https://riptutorial.com/it/csharp/topic/3569/linq-parallelo--plinq-

Capitolo 97: Linq to Objects

introduzione

LINQ to Objects fa riferimento all'utilizzo di query LINQ con qualsiasi raccolta IEnumerable.

Examples

Come LINQ to Object esegue query

Le query LINQ non vengono eseguite immediatamente. Quando si crea la query, si sta semplicemente memorizzando la query per l'esecuzione futura. Solo quando si richiede effettivamente di iterare la query è la query eseguita (ad esempio in un ciclo for, quando si chiama ToList, Count, Max, Average, First, ecc.)

Questa è considerata *un'esecuzione differita* . Ciò consente di creare la query in più passaggi, potenzialmente modificandola in base a istruzioni condizionali e quindi eseguirla in un secondo momento solo dopo aver richiesto il risultato.

Dato il codice:

```
var query = from n in numbers
            where n % 2 != 0
            select n;
```

L'esempio sopra memorizza solo la query nella variabile di `query` . Non esegue la query stessa.

L'istruzione `foreach` forza l'esecuzione della query:

```
foreach(var n in query) {
    Console.WriteLine($"Number selected {n}");
}
```

Alcuni metodi LINQ attivano anche l'esecuzione della query, `Count` , `First` , `Max` , `Average` . Restituiscono valori singoli. `ToList` e `ToArray` raccolgono i risultati e li trasformano rispettivamente in una lista o in una matrice.

Tenere presente che è possibile eseguire iterazioni sulla query più volte se si richiamano più funzioni LINQ sulla stessa query. Questo potrebbe dare risultati diversi ad ogni chiamata. Se vuoi lavorare con un solo set di dati, assicurati di salvarlo in un elenco o array.

Uso di LINQ su oggetti in C

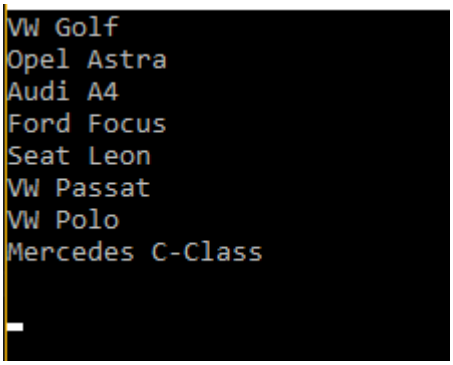
Una semplice query SELECT in Linq

```
static void Main(string[] args)
```

```
{
    string[] cars = { "VW Golf",
                      "Opel Astra",
                      "Audi A4",
                      "Ford Focus",
                      "Seat Leon",
                      "VW Passat",
                      "VW Polo",
                      "Mercedes C-Class" };

    var list = from car in cars
               select car;

    StringBuilder sb = new StringBuilder();

    foreach (string entry in list)
    {
        sb.Append(entry + "\n");
    }

    Console.WriteLine(sb.ToString());
    Console.ReadLine();
}
```

Nell'esempio sopra, una serie di stringhe (auto) viene utilizzata come una raccolta di oggetti da interrogare usando LINQ. In una query LINQ, la clausola from viene prima di tutto per introdurre l'origine dati (automobili) e la variabile range (auto). Quando viene eseguita la query, la variabile di intervallo servirà come riferimento per ciascun elemento successivo nelle automobili. Poiché il compilatore può inferire il tipo di auto, non è necessario specificarlo esplicitamente

Quando il codice sopra è compilato ed eseguito, produce il seguente risultato:

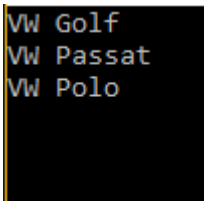


SELEZIONA con una clausola WHERE

```
var list = from car in cars
           where car.Contains("VW")
           select car;
```

La clausola WHERE viene utilizzata per interrogare l'array di stringhe (macchine) per trovare e restituire un sottoinsieme di array che soddisfi la clausola WHERE.

Quando il codice sopra è compilato ed eseguito, produce il seguente risultato:

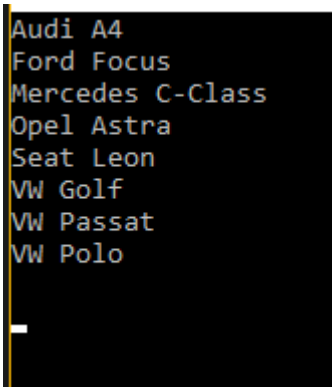


Generazione di una lista ordinata

```
var list = from car in cars
           orderby car ascending
           select car;
```

A volte è utile ordinare i dati restituiti. La clausola orderby causerà l'ordinamento degli elementi in base al confronto predefinito per il tipo da ordinare.

Quando il codice sopra è compilato ed eseguito, produce il seguente risultato:



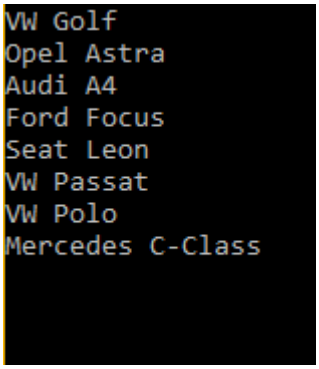
Lavorare con un tipo personalizzato

In questo esempio, un elenco tipizzato viene creato, popolato e quindi interrogato

```
public class Car
{
    public String Name { get; private set; }
    public int UnitsSold { get; private set; }

    public Car(string name, int unitsSold)
    {
        Name = name;
        UnitsSold = unitsSold;
    }
}

class Program
{
    static void Main(string[] args)
    {

        var car1 = new Car("VW Golf", 270952);
        var car2 = new Car("Opel Astra", 56079);
        var car3 = new Car("Audi A4", 52493);
        var car4 = new Car("Ford Focus", 51677);
        var car5 = new Car("Seat Leon", 42125);
        var car6 = new Car("VW Passat", 97586);
```

```
        var car7 = new Car("VW Polo", 69867);
        var car8 = new Car("Mercedes C-Class", 67549);

        var cars = new List<Car> {
            car1, car2, car3, car4, car5, car6, car7, car8 };
        var list = from car in cars
                   select car.Name;

        foreach (var entry in list)
        {
            Console.WriteLine(entry);
        }
        Console.ReadLine();
    }
}
```

Quando il codice sopra è compilato ed eseguito, produce il seguente risultato:

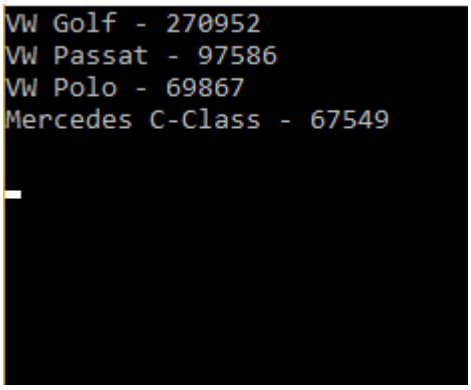


Fino ad ora gli esempi non sembrano sorprendenti in quanto si può semplicemente ripetere l'array per fare praticamente lo stesso. Tuttavia, con i pochi esempi qui sotto puoi vedere come creare query più complesse con LINQ to Objects e ottenere di più con molto meno codice.

Nell'esempio seguente possiamo selezionare le auto che sono state vendute oltre 60000 unità e ordinarle sul numero di unità vendute:

```
var list = from car in cars
           where car.UnitsSold > 60000
           orderby car.UnitsSold descending
           select car;

StringBuilder sb = new StringBuilder();

foreach (var entry in list)
{
    sb.AppendLine($"{entry.Name} - {entry.UnitsSold}");
}
Console.WriteLine(sb.ToString());
```

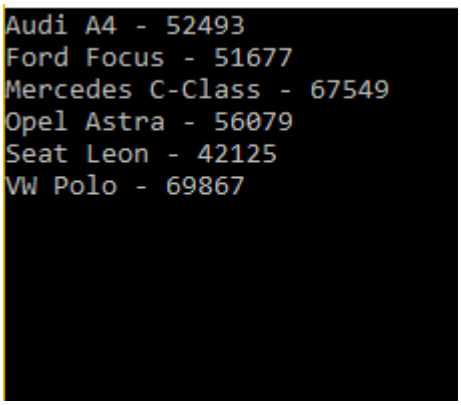
Quando il codice sopra è compilato ed eseguito, produce il seguente risultato:



Nell'esempio qui sotto possiamo selezionare le auto che hanno venduto un numero dispari di unità e ordinarle alfabeticamente sopra il suo nome:

```
var list = from car in cars
           where car.UnitsSold % 2 != 0
           orderby car.Name ascending
           select car;
```

Quando il codice sopra è compilato ed eseguito, produce il seguente risultato:



Leggi Linq to Objects online: https://riptutorial.com/it/csharp/topic/9405/linq-to-objects

Capitolo 98: Lock Statement

Sintassi

- lock (obj) {}

Osservazioni

Utilizzando l'istruzione `lock` è possibile controllare l'accesso di diversi thread al codice all'interno del blocco di codice. Viene comunemente usato per prevenire condizioni di gara, ad esempio più thread che leggono e rimuovono oggetti da una collezione. Poiché il blocco delle forze thread in attesa di altri thread per uscire da un blocco di codice, può causare ritardi che potrebbero essere risolti con altri metodi di sincronizzazione.

MSDN

> La parola chiave lock contrassegna un blocco di istruzioni come sezione critica ottenendo il blocco di esclusione reciproca per un dato oggetto, eseguendo un'istruzione e quindi rilasciando il blocco.
>
> La parola chiave lock assicura che un thread non entri in una sezione critica del codice mentre un altro thread si trova nella sezione critica. Se un altro thread tenta di inserire un codice bloccato, attenderà, bloccherà, fino a quando l'oggetto non verrà rilasciato.
>
> È consigliabile definire un oggetto **privato** da bloccare o una variabile oggetto **statica privata** per proteggere i dati comuni a tutte le istanze.

In C # 5.0 e versioni successive, l'istruzione `lock` equivale a:

```
bool lockTaken = false;
try
{
    System.Threading.Monitor.Enter(refObject, ref lockTaken);
    // code
}
finally
{
    if (lockTaken)
        System.Threading.Monitor.Exit(refObject);
}
```

Per C # 4.0 e precedenti, l'istruzione `lock` equivale a:

```
System.Threading.Monitor.Enter(refObject);
try
{
    // code
}
```

```
finally
{
     System.Threading.Monitor.Exit(refObject);
}
```

Examples

Uso semplice

L'uso comune del `lock` è una sezione critica.

Nel seguente esempio si suppone che `ReserveRoom` venga chiamato da diversi thread. La sincronizzazione con il `lock` è il modo più semplice per prevenire le condizioni di gara qui. Il corpo del metodo è circondato da un `lock` che garantisce che due o più thread non possano eseguirlo contemporaneamente.

```
public class Hotel
{
    private readonly object _roomLock = new object();

    public void ReserveRoom(int roomNumber)
    {
        // lock keyword ensures that only one thread executes critical section at once
        // in this case, reserves a hotel room of given number
        // preventing double bookings
        lock (_roomLock)
        {
            // reserve room logic goes here
        }
    }
}
```

Se un thread raggiunge il `lock` bloccato mentre un altro thread è in esecuzione al suo interno, il primo attenderà un altro per uscire dal blocco.

> È consigliabile definire un oggetto privato da bloccare o una variabile oggetto statica privata per proteggere i dati comuni a tutte le istanze.

Eccezione di lancio in una dichiarazione di blocco

Il codice seguente rilascerà il blocco. Non ci saranno problemi L'istruzione di blocco dietro le quinte funziona come `try finally`

```
lock(locker)
{
    throw new Exception();
}
```

Più può essere visto nella specifica C # 5.0 :

Una dichiarazione di `lock` del modulo

```
lock (x) ...
```

dove `x` è un'espressione di un *tipo di riferimento* , è esattamente equivalente a

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

tranne che `x` viene valutato solo una volta.

Ritorna in una dichiarazione di blocco

Il codice seguente rilascerà il blocco.

```
lock(locker)
{
    return 5;
}
```

Per una spiegazione dettagliata, questa risposta SO è raccomandata.

Utilizzo di istanze di Object per il blocco

Quando si utilizza l'istruzione di `lock` incorporata di C # è necessaria un'istanza di qualche tipo, ma il suo stato non ha importanza. Un'istanza di `object` è perfetta per questo:

```
public class ThreadSafe {
  private static readonly object locker = new object();


  public void SomeThreadSafeMethod() {
    lock (locker) {
      // Only one thread can be here at a time.
    }
  }
}
```

NB istanze di `Type` non dovrebbero essere utilizzate per questo (nel codice sopra `typeof(ThreadSafe)`) perché le istanze di `Type` sono condivise su AppDomains e quindi l'estensione del blocco può includere codice che non dovrebbe (ad esempio se `ThreadSafe` è caricato in due AppDomain nello stesso processo quindi il blocco sulla sua istanza `Type` si bloccherebbe reciprocamente).

Anti-pattern e trucchi

Blocco su una variabile allocata allo stack / locale

Uno degli errori durante l'utilizzo del `lock` è l'utilizzo di oggetti locali come locker in una funzione. Poiché queste istanze di oggetti locali saranno diverse per ogni chiamata della funzione, il `lock` non funzionerà come previsto.

```
List<string> stringList = new List<string>();

public void AddToListNotThreadSafe(string something)
{
    // DO NOT do this, as each call to this method
    // will lock on a different instance of an Object.
    // This provides no thread safety, it only degrades performance.
    var localLock = new Object();
    lock(localLock)
    {
        stringList.Add(something);
    }
}

// Define object that can be used for thread safety in the AddToList method
readonly object classLock = new object();

public void AddToList(List<string> stringList, string something)
{
    // USE THE classLock instance field to achieve a
    // thread-safe lock before adding to stringList
    lock(classLock)
    {
        stringList.Add(something);
    }
}
```

Supponendo che il blocco limiti l'accesso all'oggetto di sincronizzazione stesso

Se un thread chiama: `lock(obj)` e un altro thread chiama `obj.ToString()` secondo thread non verrà bloccato.

```
object obj = new Object();

public void SomeMethod()
{
     lock(obj)
    {
       //do dangerous stuff
    }
 }
```

```
 //Meanwhile on other tread
 public void SomeOtherMethod()
 {
   var objInString = obj.ToString(); //this does not block
 }
```

Aspettando sottoclassi per sapere quando bloccare

A volte le classi base sono progettate in modo tale che le loro sottoclassi sono obbligate a utilizzare un blocco quando accedono a determinati campi protetti:

```
public abstract class Base
{
    protected readonly object padlock;
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public abstract void Do();
}

public class Derived1 : Base
{
    public override void Do()
    {
        lock (this.padlock)
        {
            this.list.Add("Derived1");
        }
    }
}

public class Derived2 : Base
{
    public override void Do()
    {
        this.list.Add("Derived2"); // OOPS! I forgot to lock!
    }
}
```

È molto più sicuro *incapsulare il blocco* utilizzando un metodo di modello :

```
public abstract class Base
{
    private readonly object padlock; // This is now private
    protected readonly List<string> list;

    public Base()
    {
```

```
        this.padlock = new object();
        this.list = new List<string>();
    }

    public void Do()
    {
        lock (this.padlock) {
            this.DoInternal();
        }
    }

    protected abstract void DoInternal();
}

public class Derived1 : Base
{
    protected override void DoInternal()
    {
        this.list.Add("Derived1"); // Yay! No need to lock
    }
}
```

Il blocco su una variabile ValueType in scatola non si sincronizza

Nell'esempio seguente, una variabile privata viene inserita in modo implicito in quanto viene fornita come argomento di un `object` a una funzione, aspettandosi che una risorsa di monitoraggio si blocchi a. La boxe si verifica appena prima di chiamare la funzione IncInSync, quindi l'istanza boxed corrisponde a un oggetto heap diverso ogni volta che viene chiamata la funzione.

```
public int Count { get; private set; }

private readonly int counterLock = 1;

public void Inc()
{
    IncInSync(counterLock);
}

private void IncInSync(object monitorResource)
{
    lock (monitorResource)
    {
        Count++;
    }
}
```

La boxe si verifica nella funzione `Inc` :

```
BulemicCounter.Inc:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  ldarg.0
IL_0003:  ldfld       UserQuery+BulemicCounter.counterLock
```

```
IL_0008:  box         System.Int32**
IL_000D:  call        UserQuery+BulemicCounter.IncInSync
IL_0012:  nop
IL_0013:  ret
```

Ciò non significa che non sia possibile utilizzare ValueType in scatola per il blocco del monitor:

```
private readonly object counterLock = 1;
```

Ora la boxe si verifica nel costruttore, che va bene per il blocco:

```
IL_0001:  ldc.i4.1
IL_0002:  box         System.Int32
IL_0007:  stfld       UserQuery+BulemicCounter.counterLock
```

Utilizzare i blocchi inutilmente quando esiste un'alternativa più sicura

Un modello molto comune consiste nell'utilizzare un `List` o un `Dictionary` privato in una classe thread-safe e bloccarli ogni volta che si accede:

```
public class Cache
{
    private readonly object padlock;
    private readonly Dictionary<string, object> values;

    public WordStats()
    {
        this.padlock = new object();
        this.values = new Dictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        lock (this.padlock)
        {
            this.values.Add(key, value);
        }
    }

    /* rest of class omitted */
}
```

Se ci sono più metodi per accedere al dizionario dei `values` , il codice può diventare molto lungo e, soprattutto, bloccare tutto il tempo oscura il suo *intento* . Il blocco è anche molto facile da dimenticare e la mancanza di un blocco adeguato può causare errori molto difficili da trovare.

Usando un `ConcurrentDictionary` , possiamo evitare di bloccare completamente:

```
public class Cache
{
```

```
    private readonly ConcurrentDictionary<string, object> values;

    public WordStats()
    {
        this.values = new ConcurrentDictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        this.values.Add(key, value);
    }

    /* rest of class omitted */
}
```

L'uso di raccolte simultanee migliora anche le prestazioni, perché tutte impiegano tecniche di lock-free in una certa misura.

Leggi Lock Statement online: https://riptutorial.com/it/csharp/topic/1495/lock-statement

Capitolo 99: looping

Examples

Stili ciclici

Mentre

Il tipo di loop più banale. Unico inconveniente è che non esiste alcun indizio intrinseco per sapere dove ti trovi nel ciclo.

```
/// loop while the condition satisfies
while(condition)
{
    /// do something
}
```

Fare

Simile a `while` , ma la condizione viene valutata alla fine del ciclo anziché all'inizio. Ciò si traduce nell'esecuzione dei loop almeno una volta.

```
do
{
    /// do something
} while(condition) /// loop while the condition satisfies
```

Per

Un altro stile di ciclo insignificante. Mentre il ciclo di un indice (`i`) viene aumentato e puoi usarlo. Viene solitamente utilizzato per la gestione di array.

```
for ( int i = 0; i < array.Count; i++ )
{
    var currentItem = array[i];
    /// do something with "currentItem"
}
```

Per ciascuno

Modo modernizzato per il looping di oggetti `IEnumarable` . Meno male che non devi pensare all'indice dell'articolo o al numero di articoli dell'elenco.

```
foreach ( var item in someList )
{
    /// do something with "item"
}
```

Metodo Foreach

Mentre gli altri stili vengono utilizzati per selezionare o aggiornare gli elementi nelle raccolte, questo stile viene solitamente utilizzato per *chiamare* immediatamente *un metodo* per tutti gli elementi di una raccolta.

```
list.ForEach(item => item.DoSomething());

// or
list.ForEach(item => DoSomething(item));

// or using a method group
list.ForEach(Console.WriteLine);

// using an array
Array.ForEach(myArray, Console.WriteLine);
```

E 'importante notare che questo metodo in disponibile solo su `List<T>` casi e come un metodo statico su `Array` - **non** è parte di Linq.

Linq Parallelo Foreach

Proprio come Linq Foreach, eccetto questo fa il lavoro in modo parallelo. Significa che tutti gli elementi della collezione eseguiranno l'azione data contemporaneamente, contemporaneamente.

```
collection.AsParallel().ForAll(item => item.DoSomething());

/// or
collection.AsParallel().ForAll(item => DoSomething(item));
```

rompere

A volte la condizione del ciclo dovrebbe essere controllata nel mezzo del ciclo. Il primo è probabilmente più elegante del secondo:

```
for (;;)
{
    // precondition code that can change the value of should_end_loop expression

    if (should_end_loop)
        break;

    // do something
}
```

Alternativa:

```
bool endLoop = false;
for (; !endLoop;)
{
    // precondition code that can set endLoop flag

    if (!endLoop)
    {
        // do something
```

```
    }
}
```

Nota: nei cicli nidificati e / o `switch` necessario utilizzare più di una semplice `break` .

Ciclo Foreach

foreach eseguirà iterazioni su qualsiasi oggetto di una classe che implementa `IEnumerable` (prendi nota che `IEnumerable<T>` eredita da esso). Tali oggetti includono alcuni built-in, ma non limitano a: `List<T>` , `T[]` (array di qualsiasi tipo), `Dictionary<TKey, TSource>` , nonché interfacce come `IQueryable` e `ICollection` , ecc.

sintassi

```
foreach(ItemType itemVariable in enumerableObject)
    statement;
```

osservazioni

1. Il tipo `ItemType` non ha bisogno di corrispondere al tipo preciso degli articoli, deve solo essere assegnabile dal tipo di elementi
2. Invece di `ItemType` , in alternativa si può usare `var ItemType` il tipo di item da enumerableObject esaminando l'argomento generico dell'implementazione `IEnumerable`
3. L'istruzione può essere un blocco, una singola istruzione o anche un'istruzione vuota (`;`)
4. Se `enumerableObject` non sta implementando `IEnumerable` , il codice non verrà compilato
5. Durante ogni iterazione, l'oggetto corrente viene castato su `ItemType` (anche se questo non è specificato ma inferito dal compilatore tramite `var`) e se l'elemento non può essere lanciato verrà lanciata una `InvalidCastException` .

Considera questo esempio:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
foreach(var name in list)
{
    Console.WriteLine("Hello " + name);
}
```

è equivalente a:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
IEnumerator enumerator;
try
{
    enumerator = list.GetEnumerator();
    while(enumerator.MoveNext())
    {
        string name = (string)enumerator.Current;
```

```
        Console.WriteLine("Hello " + name);
    }
}
finally
{
    if (enumerator != null)
        enumerator.Dispose();
}
```

Mentre loop

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

Produzione:

0
1
2
3
4

IEnumeratori possono essere iterati con un ciclo while:

```
// Call a custom method that takes a count, and returns an IEnumerator for a list
// of strings with the names of theh largest city metro areas.
IEnumerator<string> largestMetroAreas = GetLargestMetroAreas(4);

while (largestMetroAreas.MoveNext())
{
    Console.WriteLine(largestMetroAreas.Current);
}
```

Uscita di esempio:

Tokyo / Yokohama
Metropolitana di New York
San Paolo
Seul / Incheon

Per Loop

A For Loop è ottimo per fare le cose in un certo periodo di tempo. È come un ciclo While, ma l'incremento è incluso nella condizione.

A For Loop è impostato in questo modo:

```
for (Initialization; Condition; Increment)
```

```
{
    // Code
}
```

Inizializzazione: crea una nuova variabile locale che può essere utilizzata solo nel ciclo.
Condizione: il ciclo viene eseguito solo se la condizione è vera.
Incremento - Come cambia la variabile ogni volta che viene eseguito il ciclo.

Un esempio:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Produzione:

0
1
2
3
4

Puoi anche lasciare spazi nel For Loop, ma devi avere tutti i punti e virgola per farlo funzionare.

```
int input = Console.ReadLine();

for ( ; input < 10; input + 2)
{
    Console.WriteLine(input);
}
```

Uscita per 3:

3
5
7
9
11

Do - While Loop

È simile a un ciclo `while` , tranne che verifica la condizione alla *fine* del corpo del loop. Il ciclo Do-While esegue il ciclo una volta indipendentemente dal fatto che la condizione sia vera o meno.

```
int[] numbers = new int[] { 6, 7, 8, 10 };

// Sum values from the array until we get a total that's greater than 10,
// or until we run out of values.
int sum = 0;
```

```
int i = 0;
do
{
    sum += numbers[i];
    i++;
} while (sum <= 10 && i < numbers.Length);

System.Console.WriteLine(sum); // 13
```

Anelli nidificati

```
// Print the multiplication table up to 5s
for (int i = 1; i <= 5; i++)
{
    for (int j = 1; j <= 5; j++)
    {
        int product = i * j;
        Console.WriteLine("{0} times {1} is {2}", i, j, product);
    }
}
```

Continua

Oltre alla `break` , c'è anche la parola chiave `continue` . Invece di interrompere completamente il ciclo, salterà semplicemente l'iterazione corrente. Potrebbe essere utile se non vuoi che un codice venga eseguito se è impostato un valore particolare.

Ecco un semplice esempio:

```
for (int i = 1; i <= 10; i++)
{
    if (i < 9)
        continue;

    Console.WriteLine(i);
}
```

Risulterà in:

```
9
10
```

Nota: `Continue` è spesso utile nei cicli while o do-while. For-loops, con condizioni di uscita ben definite, potrebbe non essere di beneficio.

Leggi looping online: https://riptutorial.com/it/csharp/topic/2064/looping

Capitolo 100: Manipolazione delle stringhe

Examples

Modifica del caso di caratteri all'interno di una stringa

La classe `System.String` supporta un numero di metodi per la conversione tra caratteri maiuscoli e minuscoli in una stringa.

- `System.String.ToLowerInvariant` viene utilizzato per restituire un oggetto String convertito in lettere minuscole.
- `System.String.ToUpperInvariant` viene utilizzato per restituire un oggetto String convertito in maiuscolo.

Nota: la ragione per utilizzare le versioni *invarianti* di questi metodi è impedire la produzione di lettere impreviste specifiche della cultura. Questo è spiegato qui in dettaglio .

Esempio:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Si noti che è *possibile* scegliere di specificare una **cultura** specifica durante la conversione in lettere minuscole e maiuscole utilizzando i metodi String.ToLower (CultureInfo) e String.ToUpper (CultureInfo) di conseguenza.

Trovare una stringa all'interno di una stringa

Usando `System.String.Contains` puoi scoprire se una stringa particolare esiste all'interno di una stringa. Il metodo restituisce un valore booleano, vero se la stringa esiste altrimenti false.

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the
substring
```

Utilizzando il metodo `System.String.IndexOf` , è possibile individuare la posizione iniziale di una sottostringa all'interno di una stringa esistente.
Nota che la posizione restituita è a base zero, viene restituito un valore di -1 se la sottostringa non viene trovata.

```
string s = "Hello World";
int location = s.IndexOf("ello"); // location = 1
```

Per trovare la prima posizione dalla ***fine*** di una stringa, utilizzare il metodo `System.String.LastIndexOf` :

```
string s = "Hello World";
int location = s.LastIndexOf("l"); // location = 9
```

Rimozione (ritaglio) di spazio bianco da una stringa

Il metodo `System.String.Trim` può essere utilizzato per rimuovere tutti i caratteri spazio iniziale e finale di una stringa:

```
string s = "     String with spaces at both ends      ";
s = s.Trim(); // s = "String with spaces at both ends"
```

Inoltre:

- Per rimuovere lo spazio bianco solo *dall'inizio* di una stringa, utilizzare: `System.String.TrimStart`
- Per rimuovere lo spazio bianco solo dalla *fine* di una stringa, utilizzare: `System.String.TrimEnd`

Sottostringa per estrarre parte di una stringa.

Il metodo `System.String.Substring` può essere utilizzato per estrarre una parte della stringa.

```
string s ="A portion of word that is retained";
s=str.Substring(26);  //s="retained"

s1 = s.Substring(0,5);  //s="A por"
```

Sostituzione di una stringa all'interno di una stringa

Utilizzando il metodo `System.String.Replace` , è possibile sostituire parte di una stringa con un'altra stringa.

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

Tutte le occorrenze della stringa di ricerca vengono sostituite:

```
string s = "Hello World";
s = s.Replace("l", "L"); // s = "HeLLo WorLD"
```

`String.Replace` può anche essere utilizzato per *rimuovere* parte di una stringa, specificando una stringa vuota come valore di sostituzione:

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Divisione di una stringa utilizzando un delimitatore

Utilizzare il metodo `System.String.Split` per restituire una matrice di stringhe contenente

sottostringhe della stringa originale, divisa in base a un delimitatore specificato:

```
string sentence = "One Two Three Four";
string[] stringArray = sentence.Split(' ');

foreach (string word in stringArray)
{
    Console.WriteLine(word);
}
```

Produzione:

Uno
Due
Tre
quattro

Concatena una serie di stringhe in una singola stringa

Il metodo `System.String.Join` consente di concatenare tutti gli elementi in una matrice di stringhe, utilizzando un separatore specificato tra ciascun elemento:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

Concatenazione di stringhe

String La concatenazione può essere eseguita utilizzando il metodo `System.String.Concat` o (molto più semplice) utilizzando l'operatore `+` :

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

Leggi Manipolazione delle stringhe online:
https://riptutorial.com/it/csharp/topic/3599/manipolazione-delle-stringhe

Capitolo 101: metodi

Examples

Dichiarazione di un metodo

Ogni metodo ha una firma univoca che consiste in un accessorio (`public` , `private` , ...), un modificatore opzionale (`abstract`), un nome e, se necessario, i parametri del metodo. Nota che il tipo di reso non fa parte della firma. Un prototipo di metodo ha il seguente aspetto:

```
AccessModifier OptionalModifier ReturnType MethodName(InputParameters)
{
    //Method body
}
```

`AccessModifier` può essere `public` , `protected` , `pirvate` o di default `internal` .

`OptionalModifier` può essere `override virtual abstract static new` o `sealed` .

`ReturnType` può essere `void` per nessun ritorno o può essere di qualsiasi tipo da quelli di base, come `int` a classi complesse.

un metodo può avere alcuni o nessun parametro di input. per impostare i parametri per un metodo, devi dichiarare ognuno come le normali dichiarazioni delle variabili (come `int a`), e per più di un parametro devi usare una virgola tra loro (come `int a, int b`).

I parametri possono avere valori predefiniti. per questo dovresti impostare un valore per il parametro (come `int a = 0`). se un parametro ha un valore predefinito, l'impostazione del valore di input è facoltativa.

Il seguente esempio di metodo restituisce la somma di due numeri interi:

```
private int Sum(int a, int b)
{
    return a + b;
}
```

Chiamare un metodo

Chiamando un metodo statico:

```
// Single argument
System.Console.WriteLine("Hello World");

// Multiple arguments
string name = "User";
System.Console.WriteLine("Hello, {0}!", name);
```

Chiamando un metodo statico e memorizzando il suo valore di ritorno:

```
string input = System.Console.ReadLine();
```

Chiamando un metodo di istanza:

```
int x = 42;
// The instance method called here is Int32.ToString()
string xAsString = x.ToString();
```

Chiamando un metodo generico

```
// Assuming a method 'T[] CreateArray<T>(int size)'
DateTime[] dates = CreateArray<DateTime>(8);
```

Parametri e argomenti

Un metodo può dichiarare un numero qualsiasi di parametri (in questo esempio, `i` , `s` e `o` sono i parametri):

```
static void DoSomething(int i, string s, object o) {
    Console.WriteLine(String.Format("i={0}, s={1}, o={2}", i, s, o));
}
```

I parametri possono essere utilizzati per passare valori in un metodo, in modo che il metodo possa funzionare con essi. Questo può essere ogni tipo di lavoro come la stampa dei valori o l'esecuzione di modifiche all'oggetto a cui fa riferimento un parametro o la memorizzazione dei valori.

Quando chiami il metodo, devi passare un valore reale per ogni parametro. A questo punto, i valori che effettivamente passano alla chiamata al metodo sono chiamati Arguments:

```
DoSomething(x, "hello", new object());
```

Tipi di reso

Un metodo non può restituire nulla (`void`) o un valore di un tipo specificato:

```
// If you don't want to return a value, use void as return type.
static void ReturnsNothing() {
    Console.WriteLine("Returns nothing");
}

// If you want to return a value, you need to specify its type.
static string ReturnsHelloWorld() {
    return "Hello World";
}
```

Se il metodo specifica un valore di ritorno, il metodo *deve* restituire un valore. A tale scopo, con il

`return` dichiarazione. Una volta che è stata raggiunta un'istruzione `return` , restituisce il valore specificato e qualsiasi codice dopo che non sarà più eseguito (le eccezioni sono `finally` blocchi, che saranno comunque eseguiti prima che il metodo ritorni).

Se il tuo metodo non restituisce nulla (`void`), puoi comunque utilizzare l'istruzione `return` senza un valore se vuoi tornare immediatamente dal metodo. Alla fine di tale metodo, tuttavia, una dichiarazione di `return` non sarebbe necessaria.

Esempi di validi `return` dichiarazioni:

```
return;
return 0;
return x * 2;
return Console.ReadLine();
```

Lanciare un'eccezione può terminare l'esecuzione del metodo senza restituire un valore. Inoltre, ci sono blocchi iteratori, in cui i valori di ritorno sono generati usando la parola chiave yield, ma quelli sono casi speciali che non verranno spiegati a questo punto.

Parametri predefiniti

È possibile utilizzare i parametri predefiniti se si desidera fornire l'opzione per omettere i parametri:

```
static void SaySomething(string what = "ehh") {
    Console.WriteLine(what);
}

static void Main() {
    // prints "hello"
    SaySomething("hello");
    // prints "ehh"
    SaySomething(); // The compiler compiles this as if we had typed SaySomething("ehh")
}
```

Quando si chiama un tale metodo e si omette un parametro per il quale viene fornito un valore predefinito, il compilatore inserisce automaticamente il valore predefinito.

Tenere presente che i parametri con i valori predefiniti devono essere scritti **dopo i** parametri senza valori predefiniti.

```
static void SaySomething(string say, string what = "ehh") {
        //Correct
        Console.WriteLine(say + what);
    }

static void SaySomethingElse(string what = "ehh", string say) {
        //Incorrect
        Console.WriteLine(say + what);
    }
```

ATTENZIONE : poiché funziona in questo modo, i valori predefiniti possono essere problematici in

alcuni casi. Se si modifica il valore predefinito di un parametro del metodo e non si ricompilano tutti i chiamanti di quel metodo, tali chiamanti continueranno a utilizzare il valore predefinito che era presente al momento della compilazione, probabilmente causando incoerenze.

Sovraccarico di metodi

Definizione: quando più metodi con lo stesso nome vengono dichiarati con parametri diversi, viene indicato come overload del metodo. L'overloading del metodo in genere rappresenta funzioni identiche nel loro scopo ma vengono scritte per accettare tipi di dati diversi come parametri.

Fattori che influenzano

- Numero di argomenti
- Tipo di argomenti
- Tipo di reso **

Considera un metodo denominato `Area` che eseguirà funzioni di calcolo, che accetta vari argomenti e restituisce il risultato.

Esempio

```
public string Area(int value1)
{
    return String.Format("Area of Square is {0}", value1 * value1);
}
```

Questo metodo accetta un argomento e restituisce una stringa, se chiamiamo il metodo con un numero intero (diciamo `5`) l'output sarà `"Area of Square is 25"` .

```
public  double Area(double value1, double value2)
{
    return value1 * value2;
}
```

Allo stesso modo, se passiamo due valori doppi a questo metodo, l'output sarà il prodotto dei due valori e sarà di tipo double. Questo può essere usato per la moltiplicazione e per trovare l'Area dei rettangoli

```
public double Area(double value1)
{
    return 3.14 * Math.Pow(value1,2);
}
```

Questo può essere usato specialmente per trovare l'area del cerchio, che accetta un doppio valore (`radius`) e restituisce un altro doppio valore come area.

Ognuno di questi metodi può essere chiamato normalmente senza conflitto - il compilatore esaminerà i parametri di ciascuna chiamata di metodo per determinare quale versione di `Area` deve essere utilizzata.

```
string squareArea = Area(2);
double rectangleArea = Area(32.0, 17.5);
double circleArea = Area(5.0); // all of these are valid and will compile.
```

** Nota che il tipo di ritorno *da solo* non può distinguere tra due metodi. Ad esempio, se avessimo due definizioni per Area con gli stessi parametri, ad esempio:

```
public string Area(double width, double height) { ... }
public double Area(double width, double height) { ... }
// This will NOT compile.
```

Se è necessario che la classe utilizzi gli stessi nomi di metodo che restituiscono valori diversi, è possibile rimuovere i problemi di ambiguità implementando un'interfaccia e definendone esplicitamente l'utilizzo.

```
public interface IAreaCalculatorString {

    public string Area(double width, double height);

}

public class AreaCalculator : IAreaCalculatorString {

    public string IAreaCalculatorString.Area(double width, double height) { ... }
    // Note that the method call now explicitly says it will be used when called through
    // the IAreaCalculatorString interface, allowing us to resolve the ambiguity.
    public double Area(double width, double height) { ... }
```

Metodo anonimo

I metodi anonimi forniscono una tecnica per passare un blocco di codice come parametro delegato. Sono metodi con un corpo, ma nessun nome.

```
delegate int IntOp(int lhs, int rhs);
```

```
class Program
{
    static void Main(string[] args)
    {
        // C# 2.0 definition
        IntOp add = delegate(int lhs, int rhs)
        {
            return lhs + rhs;
        };

        // C# 3.0 definition
        IntOp mul = (lhs, rhs) =>
        {
            return lhs * rhs;
        };

        // C# 3.0 definition - shorthand
        IntOp sub = (lhs, rhs) => lhs - rhs;
```

```
        // Calling each method
        Console.WriteLine("2 + 3 = " + add(2, 3));
        Console.WriteLine("2 * 3 = " + mul(2, 3));
        Console.WriteLine("2 - 3 = " + sub(2, 3));
    }
}
```

Diritti di accesso

```
// static: is callable on a class even when no instance of the class has been created
public static void MyMethod()

// virtual: can be called or overridden in an inherited class
public virtual  void MyMethod()

// internal: access is limited within the current assembly
internal  void MyMethod()

//private: access is limited only within the same class
private  void MyMethod()

//public: access right from every class / assembly
public void MyMethod()

//protected: access is limited to the containing class or types derived from it
protected void MyMethod()

//protected internal: access is limited to the current assembly or types derived from the
containing class.
protected internal void MyMethod()
```

Leggi metodi online: https://riptutorial.com/it/csharp/topic/60/metodi

Capitolo 102: Metodi DateTime

Examples

DateTime.Add (TimeSpan)

```
// Calculate what day of the week is 36 days from this instant.
System.DateTime today = System.DateTime.Now;
System.TimeSpan duration = new System.TimeSpan(36, 0, 0, 0);
System.DateTime answer = today.Add(duration);
System.Console.WriteLine("{0:dddd}", answer);
```

DateTime.AddDays (doppio)

Aggiungi giorni in un oggetto DateTime.

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(36);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("36 days from today: {0:dddd}", answer);
```

Puoi anche sottrarre giorni passando un valore negativo:

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(-3);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("-3 days from today: {0:dddd}", answer);
```

DateTime.AddHours (doppio)

```
double[] hours = {.08333, .16667, .25, .33333, .5, .66667, 1, 2,
                  29, 30, 31, 90, 365};
DateTime dateValue = new DateTime(2009, 3, 1, 12, 0, 0);

foreach (double hour in hours)
  Console.WriteLine("{0} + {1} hour(s) = {2}", dateValue, hour,
                    dateValue.AddHours(hour));
```

DateTime.AddMilliseconds (doppio)

```
string dateFormat = "MM/dd/yyyy hh:mm:ss.fffffff";
DateTime date1 = new DateTime(2010, 9, 8, 16, 0, 0);
Console.WriteLine("Original date: {0} ({1:N0} ticks)\n",
                  date1.ToString(dateFormat), date1.Ticks);

DateTime date2 = date1.AddMilliseconds(1);
Console.WriteLine("Second date:   {0} ({1:N0} ticks)",
                  date2.ToString(dateFormat), date2.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)\n",
```

```
                  date2 - date1, date2.Ticks - date1.Ticks);

DateTime date3 = date1.AddMilliseconds(1.5);
Console.WriteLine("Third date:    {0} ({1:N0} ticks)",
                  date3.ToString(dateFormat), date3.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)",
                  date3 - date1, date3.Ticks - date1.Ticks);
```

DateTime.Compare (DateTime t1, DateTime t2)

```
DateTime date1 = new DateTime(2009, 8, 1, 0, 0, 0);
DateTime date2 = new DateTime(2009, 8, 1, 12, 0, 0);
int result = DateTime.Compare(date1, date2);
string relationship;

if (result < 0)
    relationship = "is earlier than";
else if (result == 0)
    relationship = "is the same time as";
else relationship = "is later than";

Console.WriteLine("{0} {1} {2}", date1, relationship, date2);
```

DateTime.DaysInMonth (Int32, Int32)

```
const int July = 7;
const int Feb = 2;

int daysInJuly = System.DateTime.DaysInMonth(2001, July);
Console.WriteLine(daysInJuly);

// daysInFeb gets 28 because the year 1998 was not a leap year.
int daysInFeb = System.DateTime.DaysInMonth(1998, Feb);
Console.WriteLine(daysInFeb);

// daysInFebLeap gets 29 because the year 1996 was a leap year.
int daysInFebLeap = System.DateTime.DaysInMonth(1996, Feb);
Console.WriteLine(daysInFebLeap);
```

DateTime.AddYears (Int32)

Aggiungi anni sull'oggetto DateTime:

```
DateTime baseDate = new DateTime(2000, 2, 29);
Console.WriteLine("Base Date: {0:d}\n", baseDate);

// Show dates of previous fifteen years.
for (int ctr = -1; ctr >= -15; ctr--)
   Console.WriteLine("{0,2} year(s) ago:{1:d}",
                     Math.Abs(ctr), baseDate.AddYears(ctr));

Console.WriteLine();

// Show dates of next fifteen years.
for (int ctr = 1; ctr <= 15; ctr++)
```

```
   Console.WriteLine("{0,2} year(s) from now: {1:d}",
                     ctr, baseDate.AddYears(ctr));
```

Le funzioni pure avvisano quando si ha a che fare con DateTime

Wikipedia attualmente definisce una funzione pura come segue:

1. La funzione valuta sempre lo stesso valore di risultato dato lo stesso valore di argomento. Il valore del risultato della funzione non può dipendere da informazioni o stati nascosti che possono cambiare durante l'esecuzione del programma o tra diverse esecuzioni del programma, né può dipendere da qualsiasi ingresso esterno dai dispositivi I / O.
2. La valutazione del risultato non causa alcun effetto collaterale semanticamente osservabile o output, come la mutazione di oggetti mutabili o l'uscita ai dispositivi I / O

Come sviluppatore devi essere consapevole dei metodi puri e ti imbatterai spesso in molte aree. Uno che ho visto che morde molti sviluppatori junior sta lavorando con i metodi di classe DateTime. Molti di questi sono puri e se non si è a conoscenza di questi si può avere una sorpresa. Un esempio:

```
        DateTime sample = new DateTime(2016, 12, 25);
        sample.AddDays(1);
        Console.WriteLine(sample.ToShortDateString());
```

Dato l'esempio sopra, ci si può aspettare che il risultato stampato su console sia '26 / 12/2016 'ma in realtà si finisce con la stessa data. Questo perché AddDays è un metodo puro e non influisce sulla data originale. Per ottenere l'output previsto, è necessario modificare la chiamata AddDays al seguente:

```
        sample = sample.AddDays(1);
```

DateTime.Parse (String)

```
// Converts the string representation of a date and time to its DateTime equivalent

var dateTime = DateTime.Parse("14:23 22 Jul 2016");

Console.WriteLine(dateTime.ToString());
```

DateTime.TryParse (String, DateTime)

```
// Converts the specified string representation of a date and time to its DateTime equivalent
and returns a value that indicates whether the conversion succeeded

string[] dateTimeStrings = new []{
    "14:23 22 Jul 2016",
    "99:23 2x Jul 2016",
    "22/7/2016 14:23:00"
};

foreach(var dateTimeString in dateTimeStrings){
```

```
    DateTime dateTime;

    bool wasParsed = DateTime.TryParse(dateTimeString, out dateTime);

    string result = dateTimeString +
        (wasParsed
            ? $"was parsed to {dateTime}"
            : "can't be parsed to DateTime");

    Console.WriteLine(result);
}
```

Parse e TryParse con informazioni sulla cultura

Potresti volerlo usare durante l'analisi di DateTimes da culture diverse (lingue) , l'esempio seguente analizza la data olandese.

```
DateTime dateResult;
var dutchDateString = "31 oktober 1999 04:20";
var dutchCulture = CultureInfo.CreateSpecificCulture("nl-NL");
DateTime.TryParse(dutchDateString, dutchCulture, styles, out dateResult);
// output {31/10/1999 04:20:00}
```

Esempio di analisi:

```
DateTime.Parse(dutchDateString, dutchCulture)
// output {31/10/1999 04:20:00}
```

DateTime come inizializzatore in ciclo for

```
// This iterates through a range between two DateTimes
// with the given iterator (any of the Add methods)

DateTime start = new DateTime(2016, 01, 01);
DateTime until = new DateTime(2016, 02, 01);

// NOTICE: As the add methods return a new DateTime you have
// to overwrite dt in the iterator like dt = dt.Add()
for (DateTime dt = start; dt < until; dt = dt.AddDays(1))
{
    Console.WriteLine("Added {0} days. Resulting DateTime: {1}",
                      (dt - start).Days, dt.ToString());
}
```

L'iterazione su `TimeSpan` funziona allo stesso modo.

DateTime ToString, ToShortDateString, ToLongDateString e ToString formattati

```
using System;

public class Program
```

```
{
    public static void Main()
    {
        var date = new DateTime(2016,12,31);

        Console.WriteLine(date.ToString());        //Outputs: 12/31/2016 12:00:00 AM
        Console.WriteLine(date.ToShortDateString()); //Outputs: 12/31/2016
        Console.WriteLine(date.ToLongDateString()); //Outputs: Saturday, December 31, 2016
        Console.WriteLine(date.ToString("dd/MM/yyyy"));    //Outputs: 31/12/2016
    }
}
```

Data odierna

Per ottenere la data corrente si utilizza la proprietà `DateTime.Today` . Questo restituisce un oggetto `DateTime` con la data di oggi. Quando questo viene quindi convertito. `.ToString()` , viene eseguito di default nella località del sistema.

Per esempio:

```
Console.WriteLine(DateTime.Today);
```

Scrive la data odierna, nel formato locale sulla console.

DateTime Formating

Formattazione standard di data / ora

DateTimeFormatInfo specifica una serie di specificatori per la semplice formattazione di data e ora. Ogni specificatore corrisponde a un particolare modello di formato DateTimeFormatInfo.

```
//Create datetime
DateTime dt = new DateTime(2016,08,01,18,50,23,230);

var t = String.Format("{0:t}", dt); // "6:50 PM"                                ShortTime
var d = String.Format("{0:d}", dt); // "8/1/2016"                               ShortDate
var T = String.Format("{0:T}", dt); // "6:50:23 PM"                            LongTime
var D = String.Format("{0:D}", dt); // "Monday, August 1, 2016"                 LongDate
var f = String.Format("{0:f}", dt); // "Monday, August 1, 2016 6:50 PM"
LongDate+ShortTime
var F = String.Format("{0:F}", dt); // "Monday, August 1, 2016 6:50:23 PM"   FullDateTime
var g = String.Format("{0:g}", dt); // "8/1/2016 6:50 PM"
ShortDate+ShortTime
var G = String.Format("{0:G}", dt); // "8/1/2016 6:50:23 PM"
ShortDate+LongTime
var m = String.Format("{0:m}", dt); // "August 1"                              MonthDay
var y = String.Format("{0:y}", dt); // "August 2016"                           YearMonth
var r = String.Format("{0:r}", dt); // "SMon, 01 Aug 2016 18:50:23 GMT"      RFC1123
var s = String.Format("{0:s}", dt); // "2016-08-01T18:50:23"                  SortableDateTime
var u = String.Format("{0:u}", dt); // "2016-08-01 18:50:23Z"
UniversalSortableDateTime
```

Formattazione personalizzata di DateTime

Vi sono seguenti specificatori di formato personalizzato:

- `y` (anno)
- `M` (mese)
- `d` (giorno)
- `h` (ora 12)
- `H` (ora 24)
- `m` (minuto)
- `s` (secondo)
- `f` (seconda frazione)
- `F` (seconda frazione, gli zero finali sono tagliati)
- `t` (PM o AM)
- `z` (fuso orario).

```
var year =       String.Format("{0:y yy yyy yyyy}", dt); // "16 16 2016 2016"  year
var month =      String.Format("{0:M MM MMM MMMM}", dt); // "8 08 Aug August"  month
var day =        String.Format("{0:d dd ddd dddd}", dt); // "1 01 Mon Monday"  day
var hour =       String.Format("{0:h hh H HH}",     dt); // "6 06 18 18"       hour 12/24
var minute =     String.Format("{0:m mm}",          dt); // "50 50"            minute
var secound =    String.Format("{0:s ss}",          dt); // "23 23"            second
var fraction =   String.Format("{0:f ff fff ffff}", dt); // "2 23 230 2300"    sec.fraction
var fraction2 =  String.Format("{0:F FF FFF FFFF}", dt); // "2 23 23 23"       without zeroes
var period =     String.Format("{0:t tt}",          dt); // "P PM"             A.M. or P.M.
var zone =       String.Format("{0:z zz zzz}",      dt); // "+0 +00 +00:00"    time zone
```

È possibile utilizzare anche il separatore della data `/` (barra) e il separatore del tempo `:` (due punti).

Per esempio di codice

Per maggiori informazioni MSDN .

DateTime.ParseExact (String, String, IFormatProvider)

Converte la rappresentazione di stringa specificata di una data e un'ora al suo equivalente DateTime utilizzando il formato specificato e le informazioni sul formato specifico della cultura. Il formato della rappresentazione della stringa deve corrispondere esattamente al formato specificato.

Converti una stringa di formato specifica in equivalente DateTime

Diciamo che abbiamo una stringa DateTime specifica per la cultura `08-07-2016 11:30:12 PM` come `MM-dd-yyyy hh:mm:ss tt` e vogliamo convertirla in oggetto `DateTime` equivalente

```
string str = "08-07-2016 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "MM-dd-yyyy hh:mm:ss tt",
CultureInfo.CurrentCulture);
```

Converti una stringa di data e ora in un oggetto `DateTime` equivalente senza uno specifico formato di cultura

Diciamo che abbiamo una stringa DateTime in formato `dd-MM-yy hh:mm:ss tt` e vogliamo convertirla in oggetto `DateTime` equivalente, senza alcuna informazione sulla cultura specifica

```
string str = "17-06-16 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "dd-MM-yy hh:mm:ss tt",
CultureInfo.InvariantCulture);
```

Converti una stringa di data e ora in oggetto DateTime equivalente senza alcun formato di cultura specifico con formato diverso

Diciamo che abbiamo una stringa di date, un esempio come '23 -12-2016 'o '12 / 23/2016' e vogliamo convertirlo in oggetto `DateTime` equivalente, senza alcuna informazione sulla cultura specifica

```
   string date =  '23-12-2016' or date = 12/23/2016';
   string[] formats = new string[] {"dd-MM-yyyy","MM/dd/yyyy"}; // even can add more possible
formats.
   DateTime date = DateTime.ParseExact(date,formats,
CultureInfo.InvariantCulture,DateTimeStyles.None);
```

NOTA: `System.Globalization` deve essere aggiunto per CultureInfo Class

DateTime.TryParseExact (String, String, IFormatProvider, DateTimeStyles, DateTime)

Converte la rappresentazione di stringa specificata di una data e un'ora al suo equivalente DateTime utilizzando il formato specificato, le informazioni sul formato specifico della cultura e lo stile. Il formato della rappresentazione della stringa deve corrispondere esattamente al formato specificato. Il metodo restituisce un valore che indica se la conversione è riuscita.

Per esempio

```
CultureInfo enUS = new CultureInfo("en-US");
string dateString;
System.DateTime dateValue;
```

Date scadenti senza bandiere di stile.

```
dateString = " 5/01/2009 8:30 AM";
if (DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.None, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}


// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.AllowLeadingWhite, out
dateValue))
```

```
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Usa i formati personalizzati con M e MM.

```
dateString = "5/01/2009 09:00";
if(DateTime.TryParseExact(dateString, "M/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Analizza una stringa con informazioni sul fuso orario.

```
dateString = "05/01/2009 01:30:42 PM -05:00";
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}


// Allow a leading space in the date string.
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.AdjustToUniversal, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Analizza una stringa che rappresenta UTC.

```
dateString = "2008-06-11T16:11:20.0904778Z";
if(DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture, DateTimeStyles.None,
out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
  Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

if (DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture,
DateTimeStyles.RoundtripKind, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Uscite

```
' 5/01/2009 8:30 AM' is not in an acceptable format.
Converted ' 5/01/2009 8:30 AM' to 5/1/2009 8:30:00 AM (Unspecified).
Converted '5/01/2009 09:00' to 5/1/2009 9:00:00 AM (Unspecified).
'5/01/2009 09:00' is not in an acceptable format.
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 11:30:42 AM (Local).
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 6:30:42 PM (Utc).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 9:11:20 AM (Local).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 4:11:20 PM (Utc).
```

Leggi Metodi DateTime online: https://riptutorial.com/it/csharp/topic/1587/metodi-datetime

Capitolo 103: Metodi di estensione

Sintassi

- public static ReturnType MyExtensionMethod (target TargetType)
- public static ReturnType MyExtensionMethod (target TargetType, TArg1 arg1, ...)

Parametri

Parametro	Dettagli
Questo	Il primo parametro di un metodo di estensione deve essere sempre preceduta dalla `this` parola chiave, seguita dal identificativo con cui fare riferimento all'istanza "corrente" dell'oggetto che si sta estendendo

Osservazioni

I metodi di estensione sono zucchero sintattico che consente di invocare metodi statici su istanze di oggetti come se fossero membri del tipo stesso.

I metodi di estensione richiedono un oggetto target esplicito. Sarà necessario utilizzare `this` parola chiave per accedere al metodo all'interno del tipo esteso stesso.

I metodi di estensione devono essere dichiarati statici e devono vivere in una classe statica.

Quale spazio dei nomi?

La scelta dello spazio dei nomi per la classe del metodo di estensione è un compromesso tra visibilità e rilevabilità.

L' opzione più comunemente citata è quella di avere uno spazio dei nomi personalizzato per i metodi di estensione. Tuttavia ciò comporterà uno sforzo di comunicazione in modo che gli utenti del tuo codice sappiano che esistono i metodi di estensione e dove trovarli.

Un'alternativa è scegliere uno spazio dei nomi in modo tale che gli sviluppatori scoprano i metodi di estensione tramite Intellisense. Quindi, se si desidera estendere la classe `Foo` , è logico inserire i metodi di estensione nello stesso spazio dei nomi di `Foo` .

È importante rendersi conto che **nulla impedisce di utilizzare lo spazio dei nomi di "qualcun altro"** : quindi se si desidera estendere `IEnumerable` , è possibile aggiungere il proprio metodo di estensione nello spazio `System.Linq` nomi `System.Linq` .

Questa non è *sempre* una buona idea. Ad esempio, in un caso specifico, potresti voler estendere un tipo comune (`bool IsApproxEqualTo(this double value, double other)` per esempio), ma non avere quel 'inquinante' l'intero `System` . In questo caso è preferibile scegliere un namespace locale,

specifico.

Infine, è anche possibile inserire i metodi di estensione in *nessuno spazio dei nomi* !

Una buona domanda di riferimento: come gestisci gli spazi dei nomi dei tuoi metodi di estensione?

applicabilità

Bisogna fare attenzione quando si creano metodi di estensione per assicurarsi che siano appropriati per tutti i possibili input e non siano solo rilevanti per situazioni specifiche. Ad esempio, è possibile estendere classi di sistema come la `string` , che rende il nuovo codice disponibile per **qualsiasi** stringa. Se il codice deve eseguire una logica specifica del dominio su un formato stringa specifico del dominio, un metodo di estensione non sarebbe appropriato in quanto la sua presenza confonderebbe i chiamanti che lavorano con altre stringhe nel sistema.

Il seguente elenco contiene le funzionalità e le proprietà di base dei metodi di estensione

1. Deve essere un metodo statico.
2. Deve essere posizionato in una classe statica.
3. Utilizza la parola chiave "this" come primo parametro con un tipo in .NET e questo metodo verrà chiamato da una determinata istanza di tipo sul lato client.
4. E 'anche mostrato da VS intellisense. Quando premiamo il punto . dopo un'istanza di tipo, allora è disponibile in VS intellisense.
5. Un metodo di estensione deve essere nello stesso spazio dei nomi utilizzato o è necessario importare lo spazio dei nomi della classe mediante un'istruzione using.
6. Puoi dare qualsiasi nome per la classe che ha un metodo di estensione ma la classe dovrebbe essere statica.
7. Se si desidera aggiungere nuovi metodi a un tipo e non si dispone del codice sorgente, la soluzione consiste nell'utilizzare e implementare i metodi di estensione di quel tipo.
8. Se si creano metodi di estensione con gli stessi metodi di firma del tipo che si sta estendendo, i metodi di estensione non verranno mai chiamati.

Examples

Metodi di estensione: panoramica

I metodi di estensione sono stati introdotti in C # 3.0. I metodi di estensione estendono e aggiungono comportamenti a tipi esistenti senza creare un nuovo tipo derivato, ricompilando o modificando in altro modo il tipo originale. *Sono particolarmente utili quando non è possibile modificare la fonte di un tipo che si desidera migliorare.* I metodi di estensione possono essere creati per tipi di sistema, tipi definiti da terze parti e tipi definiti dall'utente. Il metodo di estensione può essere invocato come se fosse un metodo membro del tipo originale. Ciò consente di **concatenare il metodo** utilizzato per implementare un'interfaccia **fluida** .

Un metodo di estensione viene creato aggiungendo un **metodo statico** a una **classe statica** distinta dal tipo originale che viene esteso. La classe statica che contiene il metodo di estensione è spesso creata al solo scopo di contenere i metodi di estensione.

I metodi di estensione prendono uno speciale primo parametro che designa il tipo originale che viene esteso. Questo primo parametro è decorato con la parola chiave `this` (che costituisce un uso speciale e distinto di `this` in C # -it dovrebbe essere compreso come differente dall'uso di `this` che consente di fare riferimento ai membri dell'istanza dell'oggetto corrente).

Nell'esempio seguente, il tipo originale che è esteso è la `string` classe. `String` è stata estesa da un metodo `Shorten()` , che fornisce le funzionalità aggiuntive di accorciamento. La classe statica `StringExtensions` è stata creata per contenere il metodo di estensione. Il metodo di estensione `Shorten()` mostra che si tratta di un'estensione della `string` tramite il primo parametro appositamente contrassegnato. Per mostrare che il metodo `Shorten()` estende la `string` , il primo parametro è contrassegnato da `this` . Pertanto, la firma completa del primo parametro è `this string text` , dove `string` è il tipo originale che viene esteso e il `text` è il nome del parametro scelto.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}

class Program
{
    static void Main()
    {
        // This calls method String.ToUpper()
        var myString = "Hello World!".ToUpper();

        // This calls the extension method StringExtensions.Shorten()
        var newString = myString.Shorten(5);

        // It is worth noting that the above call is purely syntactic sugar
        // and the assignment below is functionally equivalent
        var newString2 = StringExtensions.Shorten(myString, 5);
    }
}
```

Live Demo su .NET Fiddle

L'oggetto passato come *primo argomento di un metodo di estensione* (che è accompagnato dalla parola chiave `this`) è l'istanza in cui viene chiamato il metodo di estensione.

Ad esempio, quando viene eseguito questo codice:

```
"some string".Shorten(5);
```

I valori degli argomenti sono i seguenti:

```
text: "some string"
length: 5
```

Si noti che i metodi di estensione sono utilizzabili solo se si trovano nello stesso spazio dei nomi della loro definizione, se lo spazio dei nomi viene importato in modo esplicito dal codice utilizzando il metodo di estensione o se la classe di estensione non ha spazio dei nomi. Le linee guida di .NET framework raccomandano di inserire le classi di estensione nel proprio spazio dei nomi. Tuttavia, questo può portare a problemi di scoperta.

Ciò non provoca conflitti tra i metodi di estensione e le librerie utilizzate, a meno che gli spazi dei nomi che potrebbero essere in conflitto siano esplicitamente inseriti. Ad esempio, LINQ Estensioni :

```
using System.Linq; // Allows use of extension methods from the System.Linq namespace

class Program
{
    static void Main()
    {
        var ints = new int[] {1, 2, 3, 4};

        // Call Where() extension method from the System.Linq namespace
        var even = ints.Where(x => x % 2 == 0);
    }
}
```

Live Demo su .NET Fiddle

Dal momento che C # 6.0, è anche possibile inserire una direttiva `using static` la *classe che* contiene i metodi di estensione. Ad esempio, `using static System.Linq.Enumerable;` . Ciò rende i metodi di estensione da quella particolare classe disponibile senza portare altri tipi dallo stesso spazio dei nomi in ambito.

Quando un metodo di classe con la stessa firma è disponibile, il compilatore dà la priorità alla chiamata del metodo di estensione. Per esempio:

```
class Test
{
   public void Hello()
   {
       Console.WriteLine("From Test");
   }
}

static class TestExtensions
{
    public static void Hello(this Test test)
    {
        Console.WriteLine("From extension method");
    }
}

class Program
{
    static void Main()
    {
```

```
        Test t = new Test();
        t.Hello(); // Prints "From Test"
    }
}
```

Demo dal vivo su .NET Fiddle

Notare che se ci sono due funzioni di estensione con la stessa firma e una di esse si trova nello stesso spazio dei nomi, a quella verrà assegnata una priorità. D'altra parte, se si accede a entrambi `using` , allora si verificherà un errore di compilazione con il messaggio:

> **La chiamata è ambigua tra i seguenti metodi o proprietà**

Si noti che la convenienza sintattica di chiamare un metodo di estensione tramite `originalTypeInstance.ExtensionMethod()` è una comodità opzionale. Il metodo può anche essere chiamato nel modo tradizionale, in modo che il primo parametro speciale venga utilizzato come parametro per il metodo.

Vale a dire, entrambi i seguenti lavori:

```
//Calling as though method belongs to string--it seamlessly extends string
String s = "Hello World";
s.Shorten(5);

//Calling as a traditional static method with two parameters
StringExtensions.Shorten(s, 5);
```

Utilizzo esplicito di un metodo di estensione

I metodi di estensione possono anche essere usati come normali metodi di classi statiche. Questo modo di chiamare un metodo di estensione è più dettagliato, ma è necessario in alcuni casi.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}
```

Uso:

```
var newString = StringExtensions.Shorten("Hello World", 5);
```

Quando chiamare i metodi di estensione come metodi statici

Esistono ancora scenari in cui è necessario utilizzare un metodo di estensione come metodo statico:

- Risolvere il conflitto con un metodo membro. Questo può accadere se una nuova versione di una libreria introduce un nuovo metodo membro con la stessa firma. In questo caso, il metodo membro sarà preferito dal compilatore.
- Risolvere i conflitti con un altro metodo di estensione con la stessa firma. Ciò può accadere se due librerie includono metodi di estensione e spazi dei nomi simili di entrambe le classi con metodi di estensione utilizzati nello stesso file.
- Passando il metodo di estensione come un gruppo di metodi in un parametro delegato.
- Esegui il tuo legame con `Reflection` .
- Utilizzo del metodo di estensione nella finestra Immediata in Visual Studio.

Uso statico

Se si `using static` direttiva `using static` per portare i membri statici di una classe statica in ambito globale, i metodi di estensione vengono saltati. Esempio:

```
using static OurNamespace.StringExtensions; // refers to class in previous example

// OK: extension method syntax still works.
"Hello World".Shorten(5);
// OK: static method syntax still works.
OurNamespace.StringExtensions.Shorten("Hello World", 5);
// Compile time error: extension methods can't be called as static without specifying class.
Shorten("Hello World", 5);
```

Se rimuovi `this` modificatore dal primo argomento del metodo `Shorten` , l'ultima riga verrà compilata.

Controllo nullo

I metodi di estensione sono metodi statici che si comportano come metodi di istanza. Tuttavia, a differenza di ciò che accade quando si chiama un metodo di istanza su un riferimento `null` , quando un metodo di estensione viene chiamato con un riferimento `null` , non lancia una `NullReferenceException` . Questo può essere abbastanza utile in alcuni scenari.

Ad esempio, si consideri la seguente classe statica:

```
public static class StringExtensions
{
    public static string EmptyIfNull(this string text)
    {
        return text ?? String.Empty;
    }

    public static string NullIfEmpty(this string text)
    {
        return String.Empty == text ? null : text;
    }
```

```
}
```

```
string nullString = null;
string emptyString = nullString.EmptyIfNull();// will return ""
string anotherNullString = emptyString.NullIfEmpty(); // will return null
```

Live Demo su .NET Fiddle

I metodi di estensione possono vedere solo i membri pubblici (o interni) della classe estesa

```
public class SomeClass
{
    public void DoStuff()
    {

    }

    protected void DoMagic()
    {

    }
}

public static class SomeClassExtensions
{
    public static void DoStuffWrapper(this SomeClass someInstance)
    {
        someInstance.DoStuff(); // ok
    }

    public static void DoMagicWrapper(this SomeClass someInstance)
    {
        someInstance.DoMagic(); // compilation error
    }
}
```

I metodi di estensione sono solo uno zucchero sintattico e non sono in realtà membri della classe che estendono. Ciò significa che non possono interrompere l'incapsulamento: hanno solo accesso al `public` (o quando è implementato nei campi, proprietà e metodi dello stesso assembly, `internal`).

Metodi di estensione generici

Proprio come altri metodi, i metodi di estensione possono utilizzare i generici. Per esempio:

```
static class Extensions
{
    public static bool HasMoreThanThreeElements<T>(this IEnumerable<T> enumerable)
    {
        return enumerable.Take(4).Count() > 3;
    }
}
```

e chiamandolo sarebbe come:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var hasMoreThanThreeElements = numbers.HasMoreThanThreeElements();
```

Visualizza la demo

Allo stesso modo per più argomenti di tipo:

```
public static TU GenericExt<T, TU>(this T obj)
{
     TU ret = default(TU);
     // do some stuff with obj
     return ret;
}
```

Chiamarlo sarebbe come:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var result = numbers.GenericExt<IEnumerable<int>,String>();
```

Visualizza la demo

È anche possibile creare metodi di estensione per tipi parzialmente vincolati in tipi multi generici:

```
class MyType<T1, T2>
{
}

static class Extensions
{
    public static void Example<T>(this MyType<int, T> test)
    {
    }
}
```

Chiamarlo sarebbe come:

```
MyType<int, string> t = new MyType<int, string>();
t.Example();
```

Visualizza la demo

Puoi anche specificare i vincoli di tipo con `where` :

```
public static bool IsDefault<T>(this T obj) where T : struct, IEquatable<T>
{
     return EqualityComparer<T>.Default.Equals(obj, default(T));
}
```

Chiamare il codice:

```
int number = 5;
var IsDefault = number.IsDefault();
```

Visualizza la demo

Invio di metodi di estensione in base al tipo statico

Viene utilizzato il tipo statico (in fase di compilazione) anziché il dinamico (tipo runtime) per abbinare i parametri.

```
public class Base
{
    public virtual string GetName()
    {
        return "Base";
    }
}

public class Derived : Base
{
    public override string GetName()
    {
        return "Derived";
    }
}

public static class Extensions
{
    public static string GetNameByExtension(this Base item)
    {
        return "Base";
    }

    public static string GetNameByExtension(this Derived item)
    {
        return "Derived";
    }
}

public static class Program
{
    public static void Main()
    {
        Derived derived = new Derived();
        Base @base = derived;

        // Use the instance method "GetName"
        Console.WriteLine(derived.GetName()); // Prints "Derived"
        Console.WriteLine(@base.GetName()); // Prints "Derived"

        // Use the static extension method "GetNameByExtension"
        Console.WriteLine(derived.GetNameByExtension()); // Prints "Derived"
        Console.WriteLine(@base.GetNameByExtension()); // Prints "Base"
    }
}
```

Live Demo su .NET Fiddle

Inoltre, il dispatch basato sul tipo statico non consente di chiamare un metodo di estensione su un oggetto `dynamic` :

```
public class Person
{
    public string Name { get; set; }
}

public static class ExtenionPerson
{
    public static string GetPersonName(this Person person)
    {
        return person.Name;
    }
}

dynamic person = new Person { Name = "Jon" };
var name = person.GetPersonName(); // RuntimeBinderException is thrown
```

I metodi di estensione non sono supportati dal codice dinamico.

```
static class Program
{
    static void Main()
    {
        dynamic dynamicObject = new ExpandoObject();

        string awesomeString = "Awesome";

        // Prints True
        Console.WriteLine(awesomeString.IsThisAwesome());

        dynamicObject.StringValue = awesomeString;

        // Prints True
        Console.WriteLine(StringExtensions.IsThisAwesome(dynamicObject.StringValue));

        // No compile time error or warning, but on runtime throws RuntimeBinderException
        Console.WriteLine(dynamicObject.StringValue.IsThisAwesome());
    }
}

static class StringExtensions
{
    public static bool IsThisAwesome(this string value)
    {
        return value.Equals("Awesome");
    }
}
```

> La ragione per cui [chiamare i metodi di estensione dal codice dinamico] non funziona è perché nei normali metodi di estensione del codice non dinamico funziona eseguendo una ricerca completa di tutte le classi note al compilatore per una classe statica che ha un metodo di estensione che corrisponde . La ricerca va in ordine in base alla nidificazione dello spazio dei nomi e disponibile `using` direttive in ogni spazio dei nomi.

> Ciò significa che per ottenere una chiamata al metodo di estensione dinamica risolta correttamente, in qualche modo il DLR deve sapere *in fase di esecuzione* quali sono stati tutti i nidi di spazio dei nomi e le direttive di `using` *nel codice sorgente* . Non abbiamo un meccanismo a portata di mano per codificare tutte queste informazioni nel sito di chiamata. Abbiamo preso in considerazione l'idea di inventare un meccanismo del genere, ma abbiamo deciso che era troppo costoso e ha prodotto un rischio di pianificazione eccessivo per meritarlo.

fonte

Metodi di estensione come wrapper fortemente tipizzati

I metodi di estensione possono essere utilizzati per scrivere wrapper fortemente tipizzati per oggetti simili a dizionari. Ad esempio una cache, `HttpContext.Items` in cetera ...

```
public static class CacheExtensions
{
    public static void SetUserInfo(this Cache cache, UserInfo data) =>
        cache["UserInfo"] = data;

    public static UserInfo GetUserInfo(this Cache cache) =>
        cache["UserInfo"] as UserInfo;
}
```

Questo approccio elimina la necessità di utilizzare stringhe letterali come chiavi in tutto il codebase così come la necessità di eseguire il casting sul tipo richiesto durante l'operazione di lettura. Complessivamente, crea un modo più sicuro e fortemente tipizzato di interagire con tali oggetti vagamente tipizzati come dizionari.

Metodi di estensione per concatenare

Quando un metodo di estensione restituisce un valore che ha lo stesso tipo di `this` argomento, può essere utilizzato per "concatenare" una o più chiamate di metodo con una firma compatibile. Ciò può essere utile per i tipi sealed e / o primitivi e consente la creazione di cosiddette API "fluide" se i nomi dei metodi vengono letti come il linguaggio naturale umano.

```
void Main()
{
    int result = 5.Increment().Decrement().Increment();
    // result is now 6
}

public static class IntExtensions
{
    public static int Increment(this int number) {
        return ++number;
    }

    public static int Decrement(this int number) {
        return --number;
    }
}
```

O come questo

```
void Main()
{
    int[] ints = new[] { 1, 2, 3, 4, 5, 6};
    int[] a = ints.WhereEven();
    //a is { 2, 4, 6 };
    int[] b = ints.WhereEven().WhereGreaterThan(2);
    //b is { 4, 6 };
}

public static class IntArrayExtensions
{
    public static int[] WhereEven(this int[] array)
    {
        //Enumerable.* extension methods use a fluent approach
        return array.Where(i => (i%2) == 0).ToArray();
    }

    public static int[] WhereGreaterThan(this int[] array, int value)
    {
        return array.Where(i => i > value).ToArray();
    }
}
```

Metodi di estensione in combinazione con interfacce

È molto conveniente usare i metodi di estensione con le interfacce poiché l'implementazione può essere archiviata al di fuori della classe e tutto ciò che serve per aggiungere alcune funzionalità alla classe è di decorare la classe con l'interfaccia.

```
public interface IInterface
{
   string Do()
}

public static class ExtensionMethods{
    public static string DoWith(this IInterface obj){
      //does something with IInterface instance
    }
}

public class Classy : IInterface
{
   // this is a wrapper method; you could also call DoWith() on a Classy instance directly,
   // provided you import the namespace containing the extension method
   public Do(){
       return this.DoWith();
   }
}
```

usare come:

```
 var classy = new Classy();
 classy.Do(); // will call the extension
 classy.DoWith(); // Classy implements IInterface so it can also be called this way
```

IList Esempio di metodo di estensione: confronto di 2 elenchi

È possibile utilizzare il seguente metodo di estensione per confrontare i contenuti di due istanze IList <T> dello stesso tipo.

Per impostazione predefinita, gli elementi vengono confrontati in base al loro ordine all'interno dell'elenco e gli elementi stessi, passando false al parametro `isOrdered` confronteranno solo gli elementi stessi indipendentemente dal loro ordine.

Affinché questo metodo funzioni, il tipo generico (`T`) deve sostituire i metodi `Equals` e `GetHashCode` .

Uso:

```
List<string> list1 = new List<string> {"a1", "a2", null, "a3"};
List<string> list2 = new List<string> {"a1", "a2", "a3", null};

list1.Compare(list2);//this gives false
list1.Compare(list2, false);//this gives true. they are equal when the order is disregarded
```

Metodo:

```
public static bool Compare<T>(this IList<T> list1, IList<T> list2, bool isOrdered = true)
{
    if (list1 == null && list2 == null)
        return true;
    if (list1 == null || list2 == null || list1.Count != list2.Count)
        return false;

    if (isOrdered)
    {
        for (int i = 0; i < list2.Count; i++)
        {
            var l1 = list1[i];
            var l2 = list2[i];
            if (
                 (l1 == null && l2 != null) ||
                 (l1 != null && l2 == null) ||
                 (!l1.Equals(l2)))
            {
                    return false;
            }
        }
        return true;
    }
    else
    {
        List<T> list2Copy = new List<T>(list2);
        //Can be done with Dictionary without O(n^2)
        for (int i = 0; i < list1.Count; i++)
        {
            if (!list2Copy.Remove(list1[i]))
                return false;
        }
        return true;
    }
}
```

Metodi di estensione con enumerazione

I metodi di estensione sono utili per aggiungere funzionalità alle enumerazioni.

Un uso comune è implementare un metodo di conversione.

```
public enum YesNo
{
    Yes,
    No,
}

public static class EnumExtentions
{
    public static bool ToBool(this YesNo yn)
    {
        return yn == YesNo.Yes;
    }
    public static YesNo ToYesNo(this bool yn)
    {
        return yn ? YesNo.Yes : YesNo.No;
    }
}
```

Ora puoi convertire rapidamente il tuo valore enum in un tipo diverso. In questo caso un bool.

```
bool yesNoBool = YesNo.Yes.ToBool(); // yesNoBool == true
YesNo yesNoEnum = false.ToYesNo();   // yesNoEnum == YesNo.No
```

In alternativa, i metodi di estensione possono essere utilizzati per aggiungere metodi come proprietà.

```
public enum Element
{
    Hydrogen,
    Helium,
    Lithium,
    Beryllium,
    Boron,
    Carbon,
    Nitrogen,
    Oxygen
    //Etc
}

public static class ElementExtensions
{
    public static double AtomicMass(this Element element)
    {
        switch(element)
        {
            case Element.Hydrogen:  return 1.00794;
            case Element.Helium:    return 4.002602;
            case Element.Lithium:   return 6.941;
            case Element.Beryllium: return 9.012182;
            case Element.Boron:     return 10.811;
```

```
            case Element.Carbon:    return 12.0107;
            case Element.Nitrogen:  return 14.0067;
            case Element.Oxygen:    return 15.9994;
            //Etc
        }
        return double.Nan;
    }
}

var massWater = 2*Element.Hydrogen.AtomicMass() + Element.Oxygen.AtomicMass();
```

Le estensioni e le interfacce insieme abilitano il codice DRY e le funzionalità di mixin-like

I metodi di estensione consentono di semplificare le definizioni dell'interfaccia includendo solo le funzionalità essenziali richieste nell'interfaccia stessa e consentendo di definire metodi di convenienza e sovraccarichi come metodi di estensione. Le interfacce con meno metodi sono più facili da implementare nelle nuove classi. Mantenere sovraccarichi come estensioni piuttosto che includerli nell'interfaccia consente di risparmiare direttamente dal codice copiato in ogni implementazione, aiutandoti a mantenere il tuo codice ASCIUTTO. Questo in effetti è simile al pattern di mixin che C # non supporta.

`System.Linq.Enumerable` estensioni di `System.Linq.Enumerable` a `IEnumerable<T>` sono un ottimo esempio di ciò. `IEnumerable<T>` richiede solo che la classe di implementazione implementa due metodi: `GetEnumerator()` generico e non generico `GetEnumerator()` . Ma `System.Linq.Enumerable` fornisce innumerevoli utilità utili come estensioni che consentono un consumo conciso e chiaro di `IEnumerable<T>` .

Quanto segue è un'interfaccia molto semplice con sovraccarichi di comodità forniti come estensioni.

```
public interface ITimeFormatter
{
   string Format(TimeSpan span);
}

public static class TimeFormatter
{
    // Provide an overload to *all* implementers of ITimeFormatter.
    public static string Format(
        this ITimeFormatter formatter,
        int millisecondsSpan)
        => formatter.Format(TimeSpan.FromMilliseconds(millisecondsSpan));
}

// Implementations only need to provide one method. Very easy to
// write additional implementations.
public class SecondsTimeFormatter : ITimeFormatter
{
   public string Format(TimeSpan span)
   {
       return $"{(int)span.TotalSeconds}s";
   }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var formatter = new SecondsTimeFormatter();
        // Callers get two method overloads!
        Console.WriteLine($"4500ms is rougly {formatter.Format(4500)}");
        var span = TimeSpan.FromSeconds(5);
        Console.WriteLine($"{span} is formatted as {formatter.Format(span)}");
    }
}
```

Metodi di estensione per la gestione di casi speciali

I metodi di estensione possono essere utilizzati per "nascondere" l'elaborazione di regole aziendali poco eleganti che altrimenti richiederebbero una funzione di chiamata con istruzioni if / then. Questo è simile e analogo alla gestione dei valori null con i metodi di estensione. Per esempio,

```
public static class CakeExtensions
{
    public static Cake EnsureTrueCake(this Cake cake)
    {
        //If the cake is a lie, substitute a cake from grandma, whose cakes aren't as tasty
but are known never to be lies. If the cake isn't a lie, don't do anything and return it.
        return CakeVerificationService.IsCakeLie(cake) ? GrandmasKitchen.Get1950sCake() :
cake;
    }
}
```

```
Cake myCake = Bakery.GetNextCake().EnsureTrueCake();
myMouth.Eat(myCake);//Eat the cake, confident that it is not a lie.
```

Utilizzo dei metodi di estensione con metodi statici e callback

Considera l'utilizzo di metodi di estensione come funzioni che racchiudono un altro codice, ecco un ottimo esempio che utilizza sia un metodo statico che un metodo di estensione per racchiudere il costrutto Try Catch. Crea il tuo codice Bullet Proof ...

```
using System;
using System.Diagnostics;

namespace Samples
{
    /// <summary>
    /// Wraps a try catch statement as a static helper which uses
    /// Extension methods for the exception
    /// </summary>
    public static class Bullet
    {
        /// <summary>
        /// Wrapper for Try Catch Statement
        /// </summary>
```

```
        /// <param name="code">Call back for code</param>
        /// <param name="error">Already handled and logged exception</param>
        public static void Proof(Action code, Action<Exception> error)
        {
            try
            {
                code();
            }
            catch (Exception iox)
            {
                //extension method used here
                iox.Log("BP2200-ERR-Unexpected Error");
                //callback, exception already handled and logged
                error(iox);
            }
        }
        /// <summary>
        /// Example of a logging method helper, this is the extension method
        /// </summary>
        /// <param name="error">The Exception to log</param>
        /// <param name="messageID">A unique error ID header</param>
        public static void Log(this Exception error, string messageID)
        {
            Trace.WriteLine(messageID);
            Trace.WriteLine(error.Message);
            Trace.WriteLine(error.StackTrace);
            Trace.WriteLine("");
        }
    }
    /// <summary>
    /// Shows how to use both the wrapper and extension methods.
    /// </summary>
    public class UseBulletProofing
    {
        public UseBulletProofing()
        {
            var ok = false;
            var result = DoSomething();
            if (!result.Contains("ERR"))
            {
                ok = true;
                DoSomethingElse();
            }
        }

        /// <summary>
        /// How to use Bullet Proofing in your code.
        /// </summary>
        /// <returns>A string</returns>
        public string DoSomething()
        {
            string result = string.Empty;
            //Note that the Bullet.Proof method forces this construct.
            Bullet.Proof(() =>
            {
                //this is the code callback
                result = "DST5900-INF-No Exceptions in this code";
            }, error =>
            {
                //error is the already logged and handled exception
                //determine the base result
```

```
                result = "DTS6200-ERR-An exception happened look at console log";
                if (error.Message.Contains("SomeMarker"))
                {
                    //filter the result for Something within the exception message
                    result = "DST6500-ERR-Some marker was found in the exception";
                }
            });
            return result;
        }

        /// <summary>
        /// Next step in workflow
        /// </summary>
        public void DoSomethingElse()
        {
            //Only called if no exception was thrown before
        }
    }
}
```

Metodi di estensione su interfacce

Una funzione utile dei metodi di estensione è che è possibile creare metodi comuni per un'interfaccia. Normalmente un'interfaccia non può avere implementazioni condivise, ma con i metodi di estensione che possono.

```
public interface IVehicle
{
    int MilesDriven { get; set; }
}

public static class Extensions
{
    public static int FeetDriven(this IVehicle vehicle)
    {
        return vehicle.MilesDriven * 5028;
    }
}
```

In questo esempio, il metodo `FeetDriven` può essere utilizzato su qualsiasi `IVehicle` . Questa logica in questo metodo si applicherebbe a tutti gli `IVehicle` , quindi può essere fatta in questo modo in modo che non ci debba essere un `FeetDriven` nella definizione `IVehicle` che sarebbe implementata allo stesso modo per tutti i bambini.

Utilizzo dei metodi di estensione per creare bellissime classi di mapping

Possiamo creare classi di mapper migliori con i metodi di estensione, Supponiamo di avere alcune classi DTO

```
public class UserDTO
{
       public AddressDTO Address { get; set; }
}
```

```
 public class AddressDTO
 {
        public string Name { get; set; }
 }
```

e ho bisogno di mappare a classi di modelli di vista corrispondenti

```
public class UserViewModel
{
    public AddressViewModel Address { get; set; }
}

public class AddressViewModel
{
    public string Name { get; set; }
}
```

quindi posso creare la mia classe di mapper come di seguito

```
public static class ViewModelMapper
{
      public static UserViewModel ToViewModel(this UserDTO user)
      {
            return user == null ?
                null :
                new UserViewModel()
                {
                    Address = user.Address.ToViewModel()
                    // Job = user.Job.ToViewModel(),
                    // Contact = user.Contact.ToViewModel() .. and so on
                };
      }

      public static AddressViewModel ToViewModel(this AddressDTO userAddr)
      {
            return userAddr == null ?
                null :
                new AddressViewModel()
                {
                    Name = userAddr.Name
                };
      }
}
```

Quindi finalmente posso richiamare il mio mapper come di seguito

```
    UserDTO userDTOObj = new UserDTO() {
            Address = new AddressDTO() {
                Name = "Address of the user"
            }
        };

    UserViewModel user = userDTOObj.ToViewModel(); // My DTO mapped to Viewmodel
```

Il bello qui è che tutti i metodi di mappatura hanno un nome comune (ToViewModel) e possiamo riutilizzarlo in diversi modi

Utilizzo dei metodi di estensione per creare nuovi tipi di raccolta (ad esempio DictList)

È possibile creare metodi di estensione per migliorare l'usabilità per le raccolte nidificate come un `Dictionary` con un valore di `List<T>` .

Considera i seguenti metodi di estensione:

```
public static class DictListExtensions
{
    public static void Add<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection> dict,
TKey key, TValue value)
            where TCollection : ICollection<TValue>, new()
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            list = new TCollection();
            dict.Add(key, list);
        }

        list.Add(value);
    }

    public static bool Remove<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection>
dict, TKey key, TValue value)
        where TCollection : ICollection<TValue>
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            return false;
        }

        var ret = list.Remove(value);
        if (list.Count == 0)
        {
            dict.Remove(key);
        }
        return ret;
    }
}
```

puoi usare i metodi di estensione come segue:

```
var dictList = new Dictionary<string, List<int>>();

dictList.Add("example", 5);
dictList.Add("example", 10);
dictList.Add("example", 15);

Console.WriteLine(String.Join(", ", dictList["example"])); // 5, 10, 15

dictList.Remove("example", 5);
dictList.Remove("example", 10);

Console.WriteLine(String.Join(", ", dictList["example"])); // 15
```

```
dictList.Remove("example", 15);

Console.WriteLine(dictList.ContainsKey("example")); // False
```

Visualizza la demo

Leggi Metodi di estensione online: https://riptutorial.com/it/csharp/topic/20/metodi-di-estensione

Capitolo 104: Microsoft.Exchange.WebServices

Examples

Recupera le impostazioni Fuori sede specificate dall'utente

Per prima cosa creiamo un oggetto `ExchangeManager` , dove il costruttore si connetterà ai servizi per noi. Ha anche un metodo `GetOofSettings` , che restituirà l'oggetto `OofSettings` per l'indirizzo email specificato:

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

namespace SetOutOfOffice
{
    class ExchangeManager
    {
        private ExchangeService Service;

        public ExchangeManager()
        {
            var password =
WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
            Connect("exchangeadmin", password);
        }
        private void Connect(string username, string password)
        {
            var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
            service.Credentials = new WebCredentials(username, password);
            service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

            Service = service;
        }
        private static bool RedirectionUrlValidationCallback(string redirectionUrl)
        {
            return
redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
        }
        public OofSettings GetOofSettings(string email)
        {
            return Service.GetUserOofSettings(email);
        }
    }
}
```

Possiamo ora chiamarlo altrove in questo modo:

```
var em = new ExchangeManager();
var oofSettings = em.GetOofSettings("testemail@domain.com");
```

Aggiorna impostazioni utente specifiche fuori sede

Utilizzando la classe di seguito, possiamo connetterci a Exchange e quindi impostare le impostazioni fuori sede di un utente specifico con `UpdateUserOof` :

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

class ExchangeManager
{
    private ExchangeService Service;

    public ExchangeManager()
    {
        var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
        Connect("exchangeadmin", password);
    }
    private void Connect(string username, string password)
    {
        var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
        service.Credentials = new WebCredentials(username, password);
        service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

        Service = service;
    }
    private static bool RedirectionUrlValidationCallback(string redirectionUrl)
    {
        return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
    }
    /// <summary>
    /// Updates the given user's Oof settings with the given details
    /// </summary>
    public void UpdateUserOof(int oofstate, DateTime starttime, DateTime endtime, int
externalaudience, string internalmsg, string externalmsg, string emailaddress)
    {
        var newSettings = new OofSettings
        {
            State = (OofState)oofstate,
            Duration = new TimeWindow(starttime, endtime),
            ExternalAudience = (OofExternalAudience)externalaudience,
            InternalReply = internalmsg,
            ExternalReply = externalmsg
        };

        Service.SetUserOofSettings(emailaddress, newSettings);
    }
}
```

Aggiorna le impostazioni utente con quanto segue:

```
var oofState = 1;
var startDate = new DateTime(01,08,2016);
var endDate = new DateTime(15,08,2016);
var externalAudience = 1;
var internalMessage = "I am not in the office!";
var externalMessage = "I am not in the office <strong>and neither are you!</strong>"
```

```
var theUser = "theuser@domain.com";

var em = new ExchangeManager();
em.UpdateUserOof(oofstate, startDate, endDate, externalAudience, internalMessage,
externalMessage, theUser);
```

Nota che puoi formattare i messaggi usando i tag `html` standard.

Leggi Microsoft.Exchange.WebServices online:
https://riptutorial.com/it/csharp/topic/4863/microsoft-exchange-webservices

Capitolo 105: Modelli di design creativo

Osservazioni

I modelli creazionali mirano a separare un sistema da come i suoi oggetti sono creati, composti e rappresentati. Aumentano la flessibilità del sistema in termini di cosa, chi, come e quando creare oggetti. I modelli creazionali incapsulano la conoscenza delle classi utilizzate da un sistema, ma nascondono i dettagli di come le istanze di queste classi vengono create e messe insieme. I programmatori hanno capito che la composizione di sistemi con l'ereditarietà rende questi sistemi troppo rigidi. I modelli creativi sono progettati per rompere questo stretto accoppiamento.

Examples

Singleton Pattern

Il modello Singleton è progettato per limitare la creazione di una classe esattamente a una singola istanza.

Questo modello viene utilizzato in uno scenario in cui ha senso avere solo uno di qualcosa, ad esempio:

- una singola classe che orchestra le interazioni di altri oggetti, es. Classe dirigente
- o una classe che rappresenta un'unica risorsa unica, es. Componente di registrazione

Uno dei metodi più comuni per implementare il pattern Singleton è tramite un **metodo factory** statico come `CreateInstance()` o `GetInstance()` (o una proprietà statica in C #, `Instance`), che viene quindi progettata per restituire sempre la stessa istanza.

La prima chiamata al metodo o alla proprietà crea e restituisce l'istanza Singleton. Successivamente, il metodo restituisce sempre la stessa istanza. In questo modo, c'è sempre un'istanza dell'oggetto Singleton.

Prevenire la creazione di istanze tramite `new` può essere realizzato rendendo `private`. i costruttori della classe `private`.

Ecco un tipico esempio di codice per l'implementazione di un modello Singleton in C #:

```
class Singleton
{
    // Because the _instance member is made private, the only way to get the single
    // instance is via the static Instance property below. This can also be similarly
    // achieved with a GetInstance() method instead of the property.
    private static Singleton _instance = null;

    // Making the constructor private prevents other instances from being
    // created via something like Singleton s = new Singleton(), protecting
    // against unintentional misuse.
    private Singleton()
```

```
    {
    }

    public static Singleton Instance
    {
        get
        {
            // The first call will create the one and only instance.
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            // Every call afterwards will return the single instance created above.
            return _instance;
        }
    }
}
```

Per illustrare ulteriormente questo modello, il codice seguente controlla se viene restituita un'istanza identica di Singleton quando la proprietà Instance viene chiamata più di una volta.

```
class Program
{
    static void Main(string[] args)
    {
        Singleton s1 = Singleton.Instance;
        Singleton s2 = Singleton.Instance;

        // Both Singleton objects above should now reference the same Singleton instance.
        if (Object.ReferenceEquals(s1, s2))
        {
            Console.WriteLine("Singleton is working");
        }
        else
        {
            // Otherwise, the Singleton Instance property is returning something
            // other than the unique, single instance when called.
            Console.WriteLine("Singleton is broken");
        }
    }
}
```

Nota: questa implementazione non è thread-safe.

Per vedere altri esempi, incluso come rendere questo thread-safe, visitare: Implementazione Singleton

I singleton sono concettualmente simili a un valore globale e causano problemi di progettazione e preoccupazioni simili. Per questo motivo, il pattern di Singleton è ampiamente considerato come un anti-pattern.

Visita "Cosa c'è di male in Singletons?" per maggiori informazioni sui problemi che sorgono con il loro uso.

In C #, hai la possibilità di creare una classe `static` , che rende tutti i membri statici e la classe non

può essere istanziata. Detto questo, è comune vedere le classi statiche usate al posto del modello Singleton.

Per le principali differenze tra i due, visita C # Singleton Pattern Versus Static Class .

Modello di metodo di fabbrica

Il metodo di fabbrica è uno dei modelli di progettazione creativi. È usato per affrontare il problema della creazione di oggetti senza specificare il tipo esatto di risultato. Questo documento ti insegnerà come usare correttamente il metodo DP di fabbrica.

Lascia che ti spieghi l'idea su un semplice esempio. Immagina di lavorare in una fabbrica che produce tre tipi di dispositivi: Amperometro, Voltmetro e misuratore di resistenza. Stai scrivendo un programma per un computer centrale che creerà un dispositivo selezionato, ma non conosci la decisione finale del tuo capo su cosa produrre.

Creiamo un `IDevice` interfaccia con alcune funzioni comuni che tutti i dispositivi hanno:

```
public interface IDevice
{
    int Measure();
    void TurnOff();
    void TurnOn();
}
```

Ora possiamo creare classi che rappresentano i nostri dispositivi. Queste classi devono implementare `IDevice` interfaccia `IDevice` :

```
public class AmMeter : IDevice
{
    private Random r = null;
    public AmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(-25, 60); }
    public void TurnOff() { Console.WriteLine("AmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("AmMeter turns on..."); }
}
public class OhmMeter : IDevice
{
    private Random r = null;
    public OhmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(0, 1000000); }
    public void TurnOff() { Console.WriteLine("OhmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("OhmMeter turns on..."); }
}
public class VoltMeter : IDevice
{
    private Random r = null;
    public VoltMeter()
    {
```

```
        r = new Random();
    }
    public int Measure() { return r.Next(-230, 230); }
    public void TurnOff() { Console.WriteLine("VoltMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("VoltMeter turns on..."); }
}
```

Ora dobbiamo definire il metodo di fabbrica. `DeviceFactory` classe `DeviceFactory` con il metodo statico all'interno:

```
public enum Device
{
    AM,
    VOLT,
    OHM
}
public class DeviceFactory
{
    public static IDevice CreateDevice(Device d)
    {
        switch(d)
        {
            case Device.AM: return new AmMeter();
            case Device.VOLT: return new VoltMeter();
            case Device.OHM: return new OhmMeter();
            default: return new AmMeter();
        }
    }
}
```

Grande! Proviamo il nostro codice:

```
public class Program
{
    static void Main(string[] args)
    {
        IDevice device = DeviceFactory.CreateDevice(Device.AM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();

        device = DeviceFactory.CreateDevice(Device.VOLT);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();

        device = DeviceFactory.CreateDevice(Device.OHM);
        device.TurnOn();
```

```
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();
    }
}
```

Questo è l'output di esempio che potresti vedere dopo aver eseguito questo codice:

AmMeter si accende ...

36

6

33

43

24

AmMeter accende le luci dicendo arrivederci!

VoltMeter si accende ...

102

-61

85

138

36

VoltMeter accende le luci dicendo arrivederci!

OhmMeter si accende ...

723.828

368.536

685.412

800.266

578.595

OhmMeter lampeggia luci dicendo arrivederci!

Modello costruttore

Separare la costruzione di un oggetto complesso dalla sua rappresentazione in modo che lo stesso processo di costruzione possa creare rappresentazioni diverse e fornire un alto livello di controllo sull'assemblaggio degli oggetti.

In questo esempio viene illustrato il modello di Builder in cui diversi veicoli vengono assemblati in modo graduale. Lo Shop utilizza VehicleBuilders per costruire una varietà di veicoli in una serie di passaggi sequenziali.

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

namespace GangOfFour.Builder
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Builder Design Pattern.
  /// </summary>
  public class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      VehicleBuilder builder;

      // Create shop with vehicle builders
      Shop shop = new Shop();

      // Construct and display vehicles
      builder = new ScooterBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      builder = new CarBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      builder = new MotorCycleBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Director' class
  /// </summary>
  class Shop
  {
    // Builder uses a complex series of steps
    public void Construct(VehicleBuilder vehicleBuilder)
    {
      vehicleBuilder.BuildFrame();
```

```
    vehicleBuilder.BuildEngine();
    vehicleBuilder.BuildWheels();
    vehicleBuilder.BuildDoors();
  }
}

/// <summary>
/// The 'Builder' abstract class
/// </summary>
abstract class VehicleBuilder
{
  protected Vehicle vehicle;

  // Gets vehicle instance
  public Vehicle Vehicle
  {
    get { return vehicle; }
  }

  // Abstract build methods
  public abstract void BuildFrame();
  public abstract void BuildEngine();
  public abstract void BuildWheels();
  public abstract void BuildDoors();
}

/// <summary>
/// The 'ConcreteBuilder1' class
/// </summary>
class MotorCycleBuilder : VehicleBuilder
{
  public MotorCycleBuilder()
  {
    vehicle = new Vehicle("MotorCycle");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "MotorCycle Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "500 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "2";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "0";
  }
}


/// <summary>
/// The 'ConcreteBuilder2' class
/// </summary>
```

```
class CarBuilder : VehicleBuilder
{
  public CarBuilder()
  {
    vehicle = new Vehicle("Car");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "Car Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "2500 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "4";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "4";
  }
}

/// <summary>
/// The 'ConcreteBuilder3' class
/// </summary>
class ScooterBuilder : VehicleBuilder
{
  public ScooterBuilder()
  {
    vehicle = new Vehicle("Scooter");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "Scooter Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "50 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "2";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "0";
  }
}

/// <summary>
/// The 'Product' class
```

```
  /// </summary>
  class Vehicle
  {
    private string _vehicleType;
    private Dictionary<string,string> _parts =
      new Dictionary<string,string>();

    // Constructor
    public Vehicle(string vehicleType)
    {
      this._vehicleType = vehicleType;
    }

    // Indexer
    public string this[string key]
    {
      get { return _parts[key]; }
      set { _parts[key] = value; }
    }

    public void Show()
    {
      Console.WriteLine("\n----------------------------");
      Console.WriteLine("Vehicle Type: {0}", _vehicleType);
      Console.WriteLine(" Frame : {0}", _parts["frame"]);
      Console.WriteLine(" Engine : {0}", _parts["engine"]);
      Console.WriteLine(" #Wheels: {0}", _parts["wheels"]);
      Console.WriteLine(" #Doors : {0}", _parts["doors"]);
    }
  }
}
```

Produzione

Tipo di veicolo: telaio dello scooter: telaio dello scooter
Motore: nessuno
Ruote: 2
#Più: 0

Tipo di veicolo: auto
Telaio: Car Frame
Motore: 2500 cc
Ruote: 4
#Più: 4

Tipo di veicolo: MotorCycle
Frame: MotorCycle Frame
Motore: 500 cc
Ruote: 2
#Più: 0

Modello di prototipo

Specificare il tipo di oggetti da creare utilizzando un'istanza prototipo e creare nuovi oggetti copiando questo prototipo.

In questo esempio viene illustrato il pattern Prototype in cui vengono creati nuovi oggetti Color copiando i colori preesistenti definiti dall'utente dello stesso tipo.

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

namespace GangOfFour.Prototype
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Prototype Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      ColorManager colormanager = new ColorManager();

      // Initialize with standard colors
      colormanager["red"] = new Color(255, 0, 0);
      colormanager["green"] = new Color(0, 255, 0);
      colormanager["blue"] = new Color(0, 0, 255);

      // User adds personalized colors
      colormanager["angry"] = new Color(255, 54, 0);
      colormanager["peace"] = new Color(128, 211, 128);
      colormanager["flame"] = new Color(211, 34, 20);

      // User clones selected colors
      Color color1 = colormanager["red"].Clone() as Color;
      Color color2 = colormanager["peace"].Clone() as Color;
      Color color3 = colormanager["flame"].Clone() as Color;

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Prototype' abstract class
  /// </summary>
  abstract class ColorPrototype
  {
    public abstract ColorPrototype Clone();
  }

  /// <summary>
  /// The 'ConcretePrototype' class
  /// </summary>
  class Color : ColorPrototype
  {
    private int _red;
    private int _green;
    private int _blue;
```

```
    // Constructor
    public Color(int red, int green, int blue)
    {
      this._red = red;
      this._green = green;
      this._blue = blue;
    }

    // Create a shallow copy
    public override ColorPrototype Clone()
    {
      Console.WriteLine(
        "Cloning color RGB: {0,3},{1,3},{2,3}",
        _red, _green, _blue);

      return this.MemberwiseClone() as ColorPrototype;
    }
  }

  /// <summary>
  /// Prototype manager
  /// </summary>
  class ColorManager
  {
    private Dictionary<string, ColorPrototype> _colors =
      new Dictionary<string, ColorPrototype>();

    // Indexer
    public ColorPrototype this[string key]
    {
      get { return _colors[key]; }
      set { _colors.Add(key, value); }
    }
  }
}
```

Produzione:

Clonazione del colore RGB: 255, 0, 0

Colore di clonazione RGB: 128,211,128

Clonazione del colore RGB: 211, 34, 20

Modello astratto di fabbrica

Fornire un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete.

In questo esempio viene dimostrata la creazione di diversi mondi animali per un gioco per computer che utilizza diverse fabbriche. Sebbene gli animali creati dalle fabbriche del continente siano diversi, le interazioni tra gli animali rimangono le stesse.

```
using System;
```

```
namespace GangOfFour.AbstractFactory
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Abstract Factory Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      // Create and run the African animal world
      ContinentFactory africa = new AfricaFactory();
      AnimalWorld world = new AnimalWorld(africa);
      world.RunFoodChain();

      // Create and run the American animal world
      ContinentFactory america = new AmericaFactory();
      world = new AnimalWorld(america);
      world.RunFoodChain();

      // Wait for user input
      Console.ReadKey();
    }
  }


  /// <summary>
  /// The 'AbstractFactory' abstract class
  /// </summary>
  abstract class ContinentFactory
  {
    public abstract Herbivore CreateHerbivore();
    public abstract Carnivore CreateCarnivore();
  }

  /// <summary>
  /// The 'ConcreteFactory1' class
  /// </summary>
  class AfricaFactory : ContinentFactory
  {
    public override Herbivore CreateHerbivore()
    {
      return new Wildebeest();
    }
    public override Carnivore CreateCarnivore()
    {
      return new Lion();
    }
  }

  /// <summary>
  /// The 'ConcreteFactory2' class
  /// </summary>
  class AmericaFactory : ContinentFactory
  {
    public override Herbivore CreateHerbivore()
    {
      return new Bison();
```

```
  }
  public override Carnivore CreateCarnivore()
  {
    return new Wolf();
  }
}

/// <summary>
/// The 'AbstractProductA' abstract class
/// </summary>
abstract class Herbivore
{
}

/// <summary>
/// The 'AbstractProductB' abstract class
/// </summary>
abstract class Carnivore
{
  public abstract void Eat(Herbivore h);
}

/// <summary>
/// The 'ProductA1' class
/// </summary>
class Wildebeest : Herbivore
{
}

/// <summary>
/// The 'ProductB1' class
/// </summary>
class Lion : Carnivore
{
  public override void Eat(Herbivore h)
  {
    // Eat Wildebeest
    Console.WriteLine(this.GetType().Name +
      " eats " + h.GetType().Name);
  }
}

/// <summary>
/// The 'ProductA2' class
/// </summary>
class Bison : Herbivore
{
}

/// <summary>
/// The 'ProductB2' class
/// </summary>
class Wolf : Carnivore
{
  public override void Eat(Herbivore h)
  {
    // Eat Bison
    Console.WriteLine(this.GetType().Name +
      " eats " + h.GetType().Name);
  }
}
```

```
  /// <summary>
  /// The 'Client' class
  /// </summary>
  class AnimalWorld
  {
    private Herbivore _herbivore;
    private Carnivore _carnivore;

    // Constructor
    public AnimalWorld(ContinentFactory factory)
    {
      _carnivore = factory.CreateCarnivore();
      _herbivore = factory.CreateHerbivore();
    }

    public void RunFoodChain()
    {
      _carnivore.Eat(_herbivore);
    }
  }
}
```

Produzione:

> Il leone mangia Gnu
>
> Il lupo mangia Bison

Leggi Modelli di design creativo online: https://riptutorial.com/it/csharp/topic/6654/modelli-di-design-creativo

Capitolo 106: Modelli di progettazione strutturale

introduzione

I modelli di progettazione strutturale sono schemi che descrivono come gli oggetti e le classi possono essere combinati e formare una struttura di grandi dimensioni e che facilitano la progettazione identificando un modo semplice per realizzare relazioni tra entità. Ci sono sette modelli strutturali descritti. Sono i seguenti: Adattatore, Ponte, Composito, Decoratore, Facciata, Peso mosca e Proxy

Examples

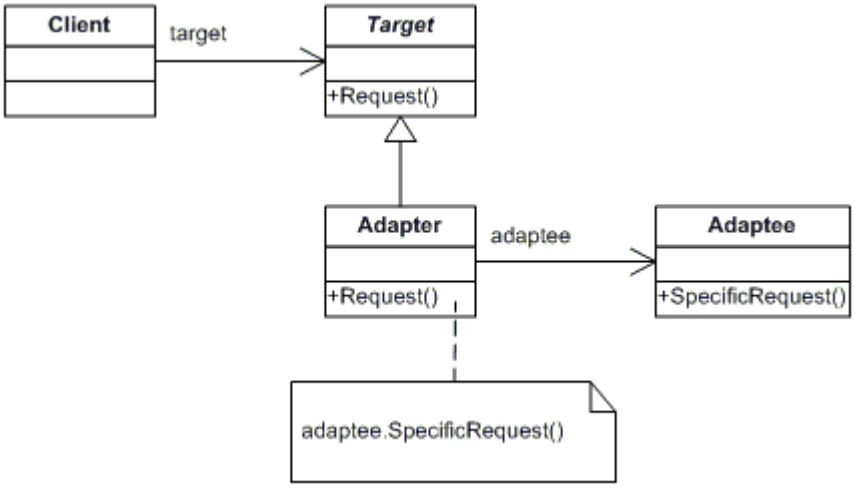
Modello di disegno dell'adattatore

> **"Adapter"** come suggerisce il nome è l'oggetto che consente a due interfacce reciprocamente incompatibili di comunicare tra loro.
>
> **Ad esempio:** se acquisti un Iphone 8 (o qualsiasi altro prodotto Apple) hai bisogno di molti adattatori. Perché l'interfaccia predefinita non supporta l'audio jac o USB. Con questi adattatori puoi usare gli auricolari con fili o puoi usare un normale cavo Ethernet. Quindi *"due interfacce reciprocamente incompatibili comunicano tra loro"* .
>
> **Quindi in termini tecnici ciò significa:** Convertire l'interfaccia di una classe in un'altra interfaccia che i clienti si aspettano. L'adattatore consente alle classi di funzionare insieme che non potrebbero altrimenti a causa di interfacce incompatibili. Le classi e gli oggetti che partecipano a questo modello sono:

Il modello dell'adattatore esclude 4 elementi

1. **ITarget:** questa è l'interfaccia che viene utilizzata dal client per ottenere funzionalità.
2. **Adaptee:** questa è la funzionalità che il cliente desidera, ma la sua interfaccia non è compatibile con il client.
3. **Cliente:** questa è la classe che vuole ottenere alcune funzionalità utilizzando il codice dell'adaparte.
4. **Adapter:** questa è la classe che implementerebbe ITarget e chiamerebbe il codice Adaptee che il client desidera chiamare.

UML



Primo esempio di codice (esempio teorico) .

```
public interface ITarget
{
    void MethodA();
}

public class Adaptee
{
    public void MethodB()
    {
        Console.WriteLine("MethodB() is called");
    }
}

public class Client
{
    private ITarget target;

    public Client(ITarget target)
    {
        this.target = target;
    }

    public void MakeRequest()
    {
        target.MethodA();
    }
}

public class Adapter : Adaptee, ITarget
{
    public void MethodA()
    {
        MethodB();
    }
}
```

Esempio di secondo codice (Real world imlementation)

```
/// <summary>
///  Interface: This is the interface which is used by the client to achieve functionality.
/// </summary>
```

```
public interface ITarget
{
    List<string> GetEmployeeList();
}

/// <summary>
/// Adaptee: This is the functionality which the client desires but its interface is not
compatible with the client.
/// </summary>
public class CompanyEmplyees
{
    public string[][] GetEmployees()
    {
        string[][] employees = new string[4][];

        employees[0] = new string[] { "100", "Deepak", "Team Leader" };
        employees[1] = new string[] { "101", "Rohit", "Developer" };
        employees[2] = new string[] { "102", "Gautam", "Developer" };
        employees[3] = new string[] { "103", "Dev", "Tester" };

        return employees;
    }
}

/// <summary>
/// Client: This is the class which wants to achieve some functionality by using the adaptee’s
code (list of employees).
/// </summary>
public class ThirdPartyBillingSystem
{
    /*
     * This class is from a thirt party and you do'n have any control over it.
     * But it requires a Emplyee list to do its work
     */

    private ITarget employeeSource;

    public ThirdPartyBillingSystem(ITarget employeeSource)
    {
        this.employeeSource = employeeSource;
    }

    public void ShowEmployeeList()
    {
        // call the clietn list in the interface
        List<string> employee = employeeSource.GetEmployeeList();

        Console.WriteLine("######### Employee List ##########");
        foreach (var item in employee)
        {
            Console.Write(item);
        }

    }
}

/// <summary>
/// Adapter: This is the class which would implement ITarget and would call the Adaptee code
which the client wants to call.
/// </summary>
public class EmployeeAdapter : CompanyEmplyees, ITarget
```

```
{
    public List<string> GetEmployeeList()
    {
        List<string> employeeList = new List<string>();
        string[][] employees = GetEmployees();
        foreach (string[] employee in employees)
        {
            employeeList.Add(employee[0]);
            employeeList.Add(",");
            employeeList.Add(employee[1]);
            employeeList.Add(",");
            employeeList.Add(employee[2]);
            employeeList.Add("\n");
        }

        return employeeList;
    }
}

///
/// Demo
///
class Programs
{
    static void Main(string[] args)
    {
        ITarget Itarget = new EmployeeAdapter();
        ThirdPartyBillingSystem client = new ThirdPartyBillingSystem(Itarget);
        client.ShowEmployeeList();
        Console.ReadKey();
    }
}
```

Quando usare

- Consentire a un sistema di utilizzare classi di un altro sistema che non è compatibile con esso.
- Consentire la comunicazione tra sistemi nuovi e già esistenti che sono indipendenti l'uno dall'altro
- Ado.Net SqlAdapter, OracleAdapter, MySqlAdapter sono il miglior esempio di Adapter Pattern.

Leggi Modelli di progettazione strutturale online: https://riptutorial.com/it/csharp/topic/9764/modelli-di-progettazione-strutturale

Capitolo 107: Modificatori di accesso

Osservazioni

Se il modificatore di accesso è omesso,

- le classi sono di default `internal`
- i metodi sono per `private`
- i getter e i setter ereditano il modificatore della proprietà, per impostazione predefinita questo è `private`

I modificatori di accesso su setter o getter di proprietà possono limitare l'accesso, non `public string someProperty {get; private set;}` : `public string someProperty {get; private set;}`

Examples

pubblico

La parola chiave `public` rende una classe (comprese le classi nidificate), proprietà, metodi o campi disponibili per ogni consumatore:

```
public class Foo()
{
    public string SomeProperty { get; set; }

    public class Baz
    {
        public int Value { get; set; }
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();
        var someValue = foo.SomeProperty;
        var myNestedInstance = new Foo.Baz();
        var otherValue = myNestedInstance.Value;
    }
}
```

privato

La parola chiave `private` contrassegna proprietà, metodi, campi e classi nidificate per l'uso solo all'interno della classe:

```
public class Foo()
{
    private string someProperty { get; set; }
```

```
    private class Baz
    {
        public string Value { get; set; }
    }

    public void Do()
    {
        var baz = new Baz { Value = 42 };
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();

        // Compile Error - not accessible due to private modifier
        var someValue = foo.someProperty;
        // Compile Error - not accessible due to private modifier
        var baz = new Foo.Baz();
    }
}
```

interno

La parola chiave internal rende una classe (incluse le classi nidificate), proprietà, metodi o campi disponibili per ogni utente nello stesso assembly:

```
internal class Foo
{
    internal string SomeProperty {get; set;}
}

internal class Bar
{
    var myInstance = new Foo();
    internal string SomeField = foo.SomeProperty;

    internal class Baz
    {
        private string blah;
        public int N { get; set; }
    }
}
```

Questo può essere interrotto per consentire a un assembly di test di accedere al codice aggiungendo il codice al file AssemblyInfo.cs:

```
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo("MyTests")]
```

protetta

La parola chiave `protected` segna il campo, le proprietà dei metodi e le classi nidificate per l'uso solo all'interno della stessa classe e delle classi derivate:

```
public class Foo()
{
    protected void SomeFooMethod()
    {
        //do something
    }

    protected class Thing
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar() : Foo
{
    private void someBarMethod()
    {
        SomeFooMethod(); // inside derived class
        var thing = new Thing(); // can use nested class
    }
}

public class Baz()
{
    private void someBazMethod()
    {
        var foo = new Foo();
        foo.SomeFooMethod(); //not accessible due to protected modifier
    }
}
```

protetto interno

La parola chiave `protected internal` contrassegna il campo, i metodi, le proprietà e le classi nidificate per l'uso all'interno dello stesso assembly o delle classi derivate in un altro assembly:

Assemblea 1

```
public class Foo
{
    public string MyPublicProperty { get; set; }
    protected internal string MyProtectedInternalProperty  { get; set; }

    protected internal class MyProtectedInternalNestedClass
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar
{
    void MyMethod1()
```

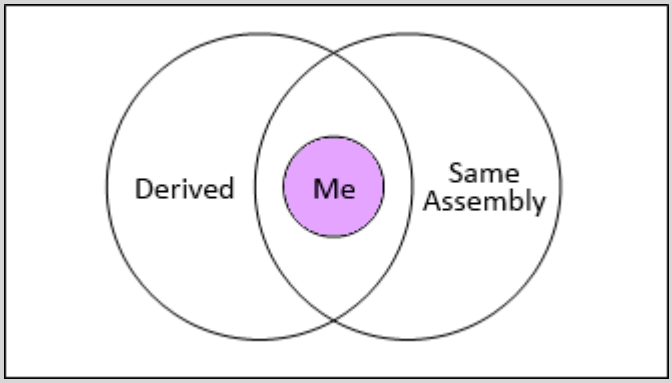
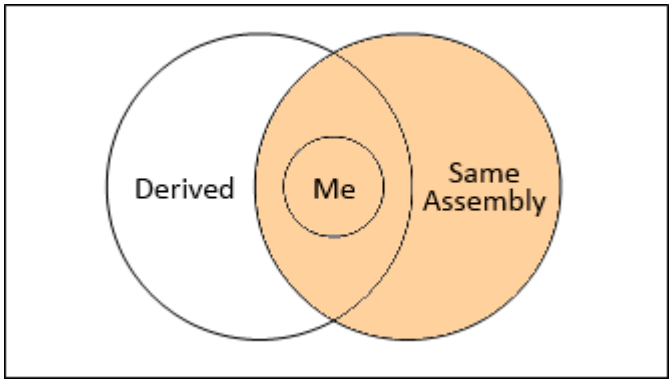
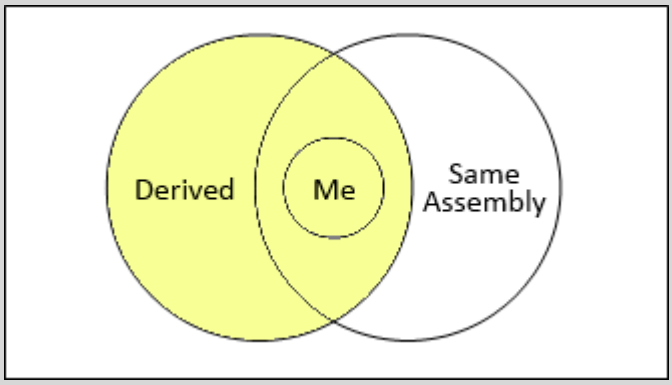
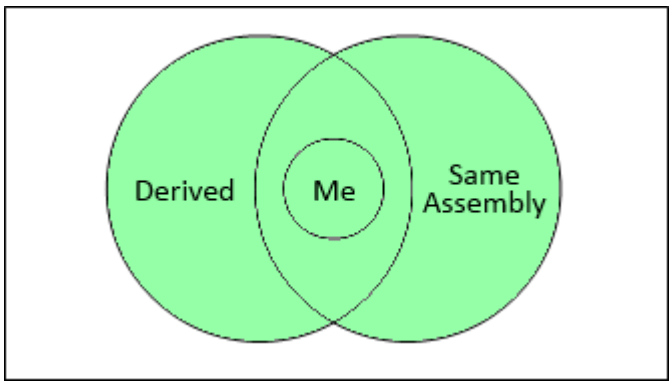
```
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}
```

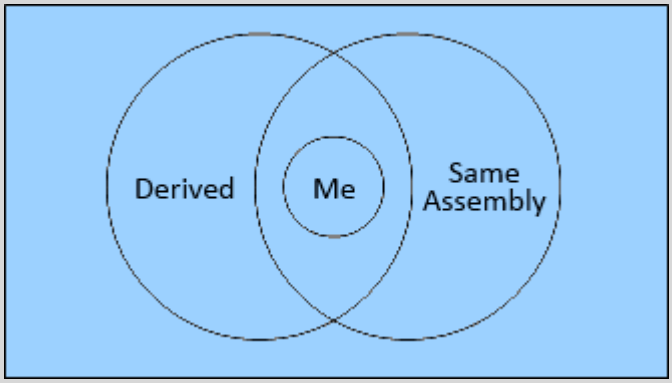
Assemblea 2

```
public class Baz : Foo
{
    void MyMethod1()
    {
        var myPublicProperty = MyPublicProperty;
        var myProtectedInternalProperty = MyProtectedInternalProperty;
        var thing = new MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }

}

public class Qux
{
    void MyMethod1()
    {
        Baz baz = new Baz();
        var myPublicProperty = baz.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = baz.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Baz.MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        //Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}
```

Access Modifiers Diagrams

Ecco tutti i modificatori di accesso nei diagrammi di Venn, da più limitanti a più accessibili:

Modificatore d'accesso	Diagramma
privato	
interno	
protetta	
protetto interno	

Modificatore d'accesso	Diagramma
pubblico	

Qui sotto puoi trovare maggiori informazioni.

Leggi Modificatori di accesso online: https://riptutorial.com/it/csharp/topic/960/modificatori-di-accesso

Capitolo 108: Networking

Sintassi

- TcpClient (host stringa, porta int);

Osservazioni

È possibile ottenere `NetworkStream` da un `TcpClient` con `client.GetStream()` e trasferirlo in uno `StreamReader/StreamWriter` per accedere ai propri metodi di lettura e scrittura asincroni.

Examples

Client di comunicazione TCP di base

Questo esempio di codice crea un client TCP, invia "Hello World" sulla connessione socket e quindi scrive la risposta del server alla console prima di chiudere la connessione.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
using (var _client = new TcpClient(host, port))
using (var _netStream = _client.GetStream())
{
    _netStream.ReadTimeout = timeout;

    // Write a message over the socket
    string message = "Hello World!";
    byte[] dataToSend = System.Text.Encoding.ASCII.GetBytes(message);
    _netStream.Write(dataToSend, 0, dataToSend.Length);

    // Read server response
    byte[] recvData = new byte[256];
    int bytes = _netStream.Read(recvData, 0, recvData.Length);
    message = System.Text.Encoding.ASCII.GetString(recvData, 0, bytes);
    Console.WriteLine(string.Format("Server: {0}", message));
};// The client and stream will close as control exits the using block (Equivilent but safer
than calling Close();
```

Scarica un file da un server web

Il download di un file da Internet è un'attività molto comune richiesta da quasi tutte le applicazioni che è probabile creare.

Per fare ciò, è possibile utilizzare la classe " System.Net.WebClient ".

L'uso più semplice di questo, usando il modello "using", è mostrato di seguito:

```
using (var webClient = new WebClient())
{
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\\file.txt");
}
```

Ciò che questo esempio fa è usare "using" per assicurarti che il tuo web client sia ripulito correttamente al termine, e semplicemente trasferisce la risorsa denominata dall'URL nel primo parametro, al file chiamato sul tuo disco rigido locale nel secondo parametro.

Il primo parametro è di tipo " System.Uri ", il secondo parametro è di tipo " System.String "

Puoi anche usare questa funzione come una forma asincrona, in modo che si spenga ed esegua il download in background, mentre la tua applicazione va avanti con qualcos'altro, usare la chiamata in questo modo è di grande importanza nelle moderne applicazioni, in quanto aiuta per mantenere la tua interfaccia utente reattiva.

Quando si utilizzano i metodi Async, è possibile collegare gestori di eventi che consentono di monitorare lo stato di avanzamento, in modo da poter ad esempio aggiornare una barra di avanzamento, ad esempio quanto segue:

```
var webClient = new WebClient())
webClient.DownloadFileCompleted += new AsyncCompletedEventHandler(Completed);
webClient.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
webClient.DownloadFileAsync("http://www.server.com/file.txt", "C:\\file.txt");
```

Un punto importante da ricordare se si usano comunque le versioni Async, e cioè "State molto attenti a usarli in una 'sintassi' che usa".

La ragione di questo è abbastanza semplice. Una volta chiamato il metodo del download, verrà restituito immediatamente. Se si dispone di questo in un blocco using, si tornerà quindi uscire da quel blocco e disporre immediatamente l'oggetto classe e quindi annullare il download in corso.

Se si utilizza il modo 'uso' per eseguire un trasferimento asincrono, assicurarsi di rimanere all'interno del blocco che lo contiene fino al completamento del trasferimento.

Client TCP asincrono

L'utilizzo di `async/await` in applicazioni C # semplifica il multi-threading. Ecco come puoi usare `async/await` insieme a un TcpClient.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
// Then get the netstream and pass it
// To our StreamWriter and StreamReader
using (var client = new TcpClient())
using (var netstream = client.GetStream())
using (var writer = new StreamWriter(netstream))
using (var reader = new StreamReader(netstream))
```

```
{
    // Asynchronsly attempt to connect to server
    await client.ConnectAsync(host, port);

    // AutoFlush the StreamWriter
    // so we don't go over the buffer
    writer.AutoFlush = true;

    // Optionally set a timeout
    netstream.ReadTimeout = timeout;

    // Write a message over the TCP Connection
    string message = "Hello World!";
    await writer.WriteLineAsync(message);

    // Read server response
    string response = await reader.ReadLineAsync();
    Console.WriteLine(string.Format($"Server: {response}"));
}
// The client and stream will close as control exits
// the using block (Equivilent but safer than calling Close();
```

Client UDP di base

Questo esempio di codice crea un client UDP quindi invia "Hello World" attraverso la rete al destinatario previsto. Un ascoltatore non deve essere attivo, poiché UDP è senza connessione e trasmetterà il messaggio indipendentemente. Una volta inviato il messaggio, il lavoro dei clienti è terminato.

```
byte[] data = Encoding.ASCII.GetBytes("Hello World");
string ipAddress = "192.168.1.141";
string sendPort = 55600;
try
{
     using (var client = new UdpClient())
     {
         IPEndPoint ep = new IPEndPoint(IPAddress.Parse(ipAddress), sendPort);
         client.Connect(ep);
         client.Send(data, data.Length);
     }
}
catch (Exception ex)
{
     Console.WriteLine(ex.ToString());
}
```

Di seguito è riportato un esempio di un listener UDP per integrare il client precedente. Si siederà e ascolterà costantemente per il traffico su una determinata porta e semplicemente scriverà quei dati sulla console. Questo esempio contiene un flag di controllo ' `done` ' che non è impostato internamente e si affida a qualcosa per impostarlo per consentire la fine del listener e l'uscita.

```
bool done = false;
int listenPort = 55600;
using(UdpClinet listener = new UdpClient(listenPort))
{
    IPEndPoint listenEndPoint = new IPEndPoint(IPAddress.Any, listenPort);
```

```
    while(!done)
    {
        byte[] receivedData = listener.Receive(ref listenPort);

        Console.WriteLine("Received broadcast message from client {0}",
listenEndPoint.ToString());

        Console.WriteLine("Decoded data is:");
        Console.WriteLine(Encoding.ASCII.GetString(receivedData)); //should be "Hello World"
sent from above client
    }
}
```

Leggi Networking online: https://riptutorial.com/it/csharp/topic/1352/networking

Capitolo 109: nome dell'operatore

introduzione

L'operatore `nameof` consente di ottenere il nome di una **variabile** , di un **tipo** o di un **membro** in forma stringa senza codificarlo come letterale.

L'operazione viene valutata in fase di compilazione, il che significa che è possibile rinominare un identificatore di riferimento, utilizzando la funzione di ridenominazione di IDE e la stringa di nome verrà aggiornata con esso.

Sintassi

- nameof (espressione)

Examples

Utilizzo di base: stampa di un nome di variabile

L'operatore `nameof` consente di ottenere il nome di una variabile, di un tipo o di un membro in forma stringa senza codificarlo come letterale. L'operazione viene valutata in fase di compilazione, il che significa che è possibile rinominare, utilizzando la funzione di ridenominazione di IDE, un identificatore di riferimento e la stringa di nome si aggiornerà con esso.

```
var myString = "String Contents";
Console.WriteLine(nameof(myString));
```

Uscirebbe

    mystring

perché il nome della variabile è "myString". Il refactoring del nome della variabile cambierebbe la stringa.

Se chiamato su un tipo di riferimento, il `nameof` operatore restituisce il nome del riferimento corrente, *non* il nome o il nome del tipo dell'oggetto sottostante. Per esempio:

```
string greeting = "Hello!";
Object mailMessageBody = greeting;

Console.WriteLine(nameof(greeting)); // Returns "greeting"
Console.WriteLine(nameof(mailMessageBody)); // Returns "mailMessageBody", NOT "greeting"!
```

Stampa di un nome parametro

Frammento

```
public void DoSomething(int paramValue)
{
    Console.WriteLine(nameof(paramValue));
}

...

int myValue = 10;
DoSomething(myValue);
```

Uscita console

paramValue

Aumentare l'evento PropertyChanged

Frammento

```
public class Person : INotifyPropertyChanged
{
    private string _address;

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public string Address
    {
        get { return _address; }
        set
        {
            if (_address == value)
            {
                return;
            }

            _address = value;
            OnPropertyChanged(nameof(Address));
        }
    }
}

...

var person = new Person();
person.PropertyChanged += (s,e) => Console.WriteLine(e.PropertyName);

person.Address = "123 Fake Street";
```

Uscita console

Indirizzo

Gestione degli eventi PropertyChanged

Frammento

```
public class BugReport : INotifyPropertyChanged
{
    public string Title { ... }
    public BugStatus Status { ... }
}

...

private void BugReport_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    var bugReport = (BugReport)sender;

    switch (e.PropertyName)
    {
        case nameof(bugReport.Title):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Title);
            break;

        case nameof(bugReport.Status):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Status);
            break;
    }
}

...

var report = new BugReport();
report.PropertyChanged += BugReport_PropertyChanged;

report.Title = "Everything is on fire and broken";
report.Status = BugStatus.ShowStopper;
```

Uscita console

Titolo cambiato in Tutto è in fiamme e rotto

Stato cambiato in ShowStopper

Applicato a un parametro di tipo generico

Frammento

```
public class SomeClass<TItem>
{
    public void PrintTypeName()
    {
        Console.WriteLine(nameof(TItem));
    }
}

...

var myClass = new SomeClass<int>();
myClass.PrintTypeName();
```

```
Console.WriteLine(nameof(SomeClass<int>));
```

Uscita console

TItem

SomeClass

Applicato agli identificatori qualificati

Frammento

```
Console.WriteLine(nameof(CompanyNamespace.MyNamespace));
Console.WriteLine(nameof(MyClass));
Console.WriteLine(nameof(MyClass.MyNestedClass));
Console.WriteLine(nameof(MyNamespace.MyClass.MyNestedClass.MyStaticProperty));
```

Uscita console

MyNamespace

La mia classe

MyNestedClass

MyStaticProperty

Argument Checking and Guard Clauses

Preferire

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException(nameof(orderLine));
        ...
    }
}
```

Al di sopra di

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException("orderLine");
        ...
    }
}
```

L'uso della funzione `nameof` semplifica il refactoring dei parametri del metodo.

Link di azione MVC fortemente tipizzati

Invece dei soliti caratteri tipografici:

```
@Html.ActionLink("Log in", "UserController", "LogIn")
```

Ora puoi creare link di azione fortemente digitati:

```
@Html.ActionLink("Log in", @typeof(UserController), @nameof(UserController.LogIn))
```

Ora, se si vuole refactoring il codice e rinominare `UserController.LogIn` metodo per `UserController.SignIn` , non è necessario preoccuparsi di ricerca di tutte le occorrenze della stringa. Il compilatore farà il lavoro.

Leggi nome dell'operatore online: https://riptutorial.com/it/csharp/topic/80/nome-dell-operatore

Capitolo 110: Nullable types

Sintassi

- `Nullable<int> i = 10;`
- int? j = 11;
- int? k = null;
- Appuntamento? DateOfBirth = DateTime.Now;
- decimale? Quantità = 1,0 m;
- bool? IsAvailable = true;
- char? Lettera = 'a';
- (genere)? variableName

Osservazioni

I tipi Nullable possono rappresentare tutti i valori di un tipo sottostante e `null` .

La sintassi `T?` è una scorciatoia per `Nullable<T>`

I valori Nullable sono in realtà oggetti `System.ValueType` , quindi possono essere inseriti in box e unbox. Inoltre, il valore `null` di un oggetto nullable non è uguale al valore `null` di un oggetto di riferimento, è solo una bandiera.

Quando un oggetto nullable boxing, il valore null viene convertito in riferimento `null` e il valore non null viene convertito in un tipo sottostante non nullable.

```
DateTime? dt = null;
var o = (object)dt;
var result = (o == null); // is true

DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var dt2 = (DateTime)dt; // correct cause o contains DateTime value
```

La seconda regola porta al codice corretto, ma paradossale:

```
DateTime? dt = new DateTime(2015, 12, 11);
var o = (object)dt;
var type = o.GetType(); // is DateTime, not Nullable<DateTime>
```

In forma breve:

```
DateTime? dt = new DateTime(2015, 12, 11);
var type = dt.GetType(); // is DateTime, not Nullable<DateTime>
```

Examples

Inizializzazione di un valore nullo

Per valori `null` :

```
Nullable<int> i = null;
```

O:

```
int? i = null;
```

O:

```
var i = (int?)null;
```

Per valori non nulli:

```
Nullable<int> i = 0;
```

O:

```
int? i = 0;
```

Controlla se un Nullable ha un valore

```
int? i = null;

if (i != null)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Che è lo stesso di:

```
if (i.HasValue)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Ottieni il valore di un tipo nullable

Dato il seguente nullable `int`

```
int? i = 10;
```

Nel caso in cui sia necessario un valore predefinito, è possibile assegnarne uno utilizzando l'operatore coalescente null , il metodo `GetValueOrDefault` o il controllo se `HasValue` int `HasValue` prima dell'assegnazione.

```
int j = i ?? 0;
int j = i.GetValueOrDefault(0);
int j = i.HasValue ? i.Value : 0;
```

Il seguente utilizzo è sempre *pericoloso* . Se `i` è null in fase di esecuzione, verrà lanciata `System.InvalidOperationException` . In fase di progettazione, se non viene impostato un valore, verrà visualizzato l'errore `Use of unassigned local variable 'i'` .

```
int j = i.Value;
```

Ottenere un valore predefinito da un valore nullo

Il metodo `.GetValueOrDefault()` restituisce un valore anche se la proprietà `.HasValue` è falsa (diversamente dalla proprietà Value, che genera un'eccezione).

```
class Program
{
    static void Main()
    {
        int? nullableExample = null;
        int result = nullableExample.GetValueOrDefault();
        Console.WriteLine(result); // will output the default value for int - 0
        int secondResult = nullableExample.GetValueOrDefault(1);
        Console.WriteLine(secondResult) // will output our specified default - 1
        int thirdResult = nullableExample ?? 1;
        Console.WriteLine(secondResult) // same as the GetValueOrDefault but a bit shorter
    }
}
```

Produzione:

```
0
1
```

Controlla se un parametro di tipo generico è un tipo nullable

```
public bool IsTypeNullable<T>()
{
    return Nullable.GetUnderlyingType( typeof(T) )!=null;
}
```

Il valore predefinito dei tipi nullable è nullo

```
public class NullableTypesExample
{
    static int? _testValue;

    public static void Main()
    {
        if(_testValue == null)
            Console.WriteLine("null");
        else
            Console.WriteLine(_testValue.ToString());
    }
}
```

Produzione:

nullo

Uso efficace del sottostante Nullable discussione

Qualsiasi tipo nullable è un tipo **generico** . E qualsiasi tipo nullable è un tipo di **valore** .

Esistono alcuni trucchi che consentono di **utilizzare in modo efficace** il risultato del metodo Nullable.GetUnderlyingType durante la creazione di codice correlato agli scopi reflection / code-generation:

```
public static class TypesHelper {
    public static bool IsNullable(this Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType);
    }
    public static bool IsNullable(this Type type, out Type underlyingType) {
        underlyingType = Nullable.GetUnderlyingType(type);
        return underlyingType != null;
    }
    public static Type GetNullable(Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType) ? type : NullableTypesCache.Get(type);
    }
    public static bool IsExactOrNullable(this Type type, Func<Type, bool> predicate) {
        Type underlyingType;
        if(IsNullable(type, out underlyingType))
            return IsExactOrNullable(underlyingType, predicate);
        return predicate(type);
    }
    public static bool IsExactOrNullable<T>(this Type type)
        where T : struct {
        return IsExactOrNullable(type, t => Equals(t, typeof(T)));
    }
}
```

L'utilizzo:

```
Type type = typeof(int).GetNullable();
Console.WriteLine(type.ToString());

if(type.IsNullable())
```

```
    Console.WriteLine("Type is nullable.");
Type underlyingType;
if(type.IsNullable(out underlyingType))
    Console.WriteLine("The underlying type is " + underlyingType.Name + ".");
if(type.IsExactOrNullable<int>())
    Console.WriteLine("Type is either exact or nullable Int32.");
if(!type.IsExactOrNullable(t => t.IsEnum))
    Console.WriteLine("Type is neither exact nor nullable enum.");
```

Produzione:

```
System.Nullable`1[System.Int32]
Type is nullable.
The underlying type is Int32.
Type is either exact or nullable Int32.
Type is neither exact nor nullable enum.
```

PS. `NullableTypesCache` è definito come segue:

```
static class NullableTypesCache {
    readonly static ConcurrentDictionary<Type, Type> cache = new ConcurrentDictionary<Type,
Type>();
    static NullableTypesCache() {
        cache.TryAdd(typeof(byte), typeof(Nullable<byte>));
        cache.TryAdd(typeof(short), typeof(Nullable<short>));
        cache.TryAdd(typeof(int), typeof(Nullable<int>));
        cache.TryAdd(typeof(long), typeof(Nullable<long>));
        cache.TryAdd(typeof(float), typeof(Nullable<float>));
        cache.TryAdd(typeof(double), typeof(Nullable<double>));
        cache.TryAdd(typeof(decimal), typeof(Nullable<decimal>));
        cache.TryAdd(typeof(sbyte), typeof(Nullable<sbyte>));
        cache.TryAdd(typeof(ushort), typeof(Nullable<ushort>));
        cache.TryAdd(typeof(uint), typeof(Nullable<uint>));
        cache.TryAdd(typeof(ulong), typeof(Nullable<ulong>));
        //...
    }
    readonly static Type NullableBase = typeof(Nullable<>);
    internal static Type Get(Type type) {
        // Try to avoid the expensive MakeGenericType method call
        return cache.GetOrAdd(type, t => NullableBase.MakeGenericType(t));
    }
}
```

Leggi Nullable types online: https://riptutorial.com/it/csharp/topic/1240/nullable-types

Capitolo 111: NullReferenceException

Examples

Spiegazione di NullReferenceException

Viene generata una `NullReferenceException` quando si tenta di accedere a un membro non statico (proprietà, metodo, campo o evento) di un oggetto di riferimento ma è nullo.

```
Car myFirstCar = new Car();
Car mySecondCar = null;
Color myFirstColor = myFirstCar.Color; // No problem as myFirstCar exists / is not null
Color mySecondColor = mySecondCar.Color; // Throws a NullReferenceException
// as mySecondCar is null and yet we try to access its color.
```

Per eseguire il debug di questa eccezione, è abbastanza semplice: sulla riga in cui viene lanciata l'eccezione, è sufficiente guardare prima di ogni ' `.` 'o' `[` ', o in rare occasioni' `(` '.

```
myGarage.CarCollection[currentIndex.Value].Color = theCarInTheStreet.Color;
```

Da dove viene la mia eccezione? O:

- `myGarage` è `null`
- `myGarage.CarCollection` è `null`
- `currentIndex` è `null`
- `myGarage.CarCollection[currentIndex.Value]` è `null`
- `theCarInTheStreet` è `null`

In modalità di debug, devi solo posizionare il cursore del mouse su ognuno di questi elementi e troverai il tuo riferimento null. Quindi, quello che devi fare è capire perché non ha un valore. La correzione dipende totalmente dall'obiettivo del tuo metodo.

Hai dimenticato di istanziarlo / inizializzarlo?

```
myGarage.CarCollection = new Car[10];
```

Dovresti fare qualcosa di diverso se l'oggetto è nullo?

```
if (myGarage == null)
{
    Console.WriteLine("Maybe you should buy a garage first!");
}
```

O forse qualcuno ti ha dato un argomento nullo, e non avrebbe dovuto:

```
if (theCarInTheStreet == null)
{
```

```
    throw new ArgumentNullException("theCarInTheStreet");
}
```

In ogni caso, ricorda che un metodo non dovrebbe mai lanciare una NullReferenceException. Se lo fa, significa che hai dimenticato di controllare qualcosa.

Leggi NullReferenceException online:
https://riptutorial.com/it/csharp/topic/2702/nullreferenceexception

Capitolo 112: O (n) Algoritmo per la rotazione circolare di un array

introduzione

Nel mio percorso di studio della programmazione ci sono stati problemi semplici ma interessanti da risolvere come esercizi. Uno di questi problemi era ruotare un array (o un'altra raccolta) di un certo valore. Qui condividerò con voi una semplice formula per farlo.

Examples

Esempio di un metodo generico che ruota un array per un dato turno

Vorrei sottolineare che ruotiamo a sinistra quando il valore di spostamento è negativo e ruotiamo a destra quando il valore è positivo.

```
    public static void Main()
    {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int shiftCount = 1;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        array = new []{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = 15;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

        array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = -1;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]

        array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = -35;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
    }

    private static void Rotate<T>(ref T[] array, int shiftCount)
    {
        T[] backupArray= new T[array.Length];

        for (int index = 0; index < array.Length; index++)
        {
            backupArray[(index + array.Length + shiftCount % array.Length) % array.Length] =
array[index];
        }
```

```
        array = backupArray;
    }
```

La cosa importante in questo codice è la formula con cui troviamo il nuovo valore di indice dopo la rotazione.

(index + array.Length + shiftCount% array.Length)% array.Length

Ecco alcune informazioni in più:

(shiftCount% array.Length) -> normalizziamo il valore di spostamento in modo che sia nella lunghezza dell'array (poiché in un array con lunghezza 10, lo spostamento 1 o 11 è la stessa cosa, lo stesso vale per -1 e -11) .

array.Length + (shiftCount% array.Length) -> questo viene fatto a causa delle rotazioni a sinistra per assicurarsi che non entriamo in un indice negativo, ma ruotarlo fino alla fine dell'array. Senza di esso per una matrice di lunghezza 10 per l'indice 0 e una rotazione -1 dovremmo inserire un numero negativo (-1) e non ottenere il valore dell'indice di rotazione reale, che è 9. (10 + (-1% 10) = 9)

index + array.Length + (shiftCount% array.Length) -> Non c'è molto da dire qui quando applichiamo la rotazione all'indice per ottenere il nuovo indice. (0 + 10 + (-1% 10) = 9)

index + array.Length + (shiftCount% array.Length)% array.Length -> la seconda normalizzazione si sta accertando che il nuovo valore di indice non vada fuori dalla matrice, ma ruoti il valore all'inizio dell'array. È per le giuste rotazioni, poiché in un array con lunghezza 10 senza esso per l'indice 9 e una rotazione 1 dovremmo andare nell'indice 10, che è al di fuori dell'array, e non ottenere il valore dell'indice di rotazione reale è 0. ((9 + 10 + (1% 10))% 10 = 0)

Leggi O (n) Algoritmo per la rotazione circolare di un array online:
https://riptutorial.com/it/csharp/topic/9770/o--n--algoritmo-per-la-rotazione-circolare-di-un-array

Capitolo 113: ObservableCollection

Examples

Inizializza ObservableCollection

`ObservableCollection` è una raccolta di tipo `T` come `List<T>` che significa che contiene oggetti di tipo `T`

Dalla documentazione leggiamo che:

> `ObservableCollection` rappresenta una raccolta di dati dinamica che fornisce notifiche quando gli articoli vengono aggiunti, rimossi o quando l'intero elenco viene aggiornato.

La differenza fondamentale rispetto alle altre raccolte è che `ObservableCollection` implementa le interfacce `INotifyCollectionChanged` e `INotifyCollectionChanged` e `INotifyPropertyChanged` immediatamente un evento di notifica quando un nuovo oggetto viene aggiunto o rimosso e quando la raccolta viene cancellata.

Ciò è particolarmente utile per connettere l'interfaccia utente e il back-end di un'applicazione senza dover scrivere codice aggiuntivo perché quando un oggetto viene aggiunto o rimosso da una raccolta osservabile, l'interfaccia utente viene automaticamente aggiornata.

Il primo passo per usarlo è includere

```
using System.Collections.ObjectModel
```

È possibile creare un'istanza vuota di una raccolta per esempio di tipo `string`

```
ObservableCollection<string> collection = new ObservableCollection<string>();
```

o un'istanza che è piena di dati

```
ObservableCollection<string> collection = new ObservableCollection<string>()
{
 "First_String", "Second_String"
};
```

Ricordare come in tutte le raccolte IList, l'indice inizia da 0 (proprietà IList.Item).

Leggi ObservableCollection online:
https://riptutorial.com/it/csharp/topic/7351/observablecollection--t-

Capitolo 114: Operatore di uguaglianza

Examples

Tipi di uguaglianza in c # e operatore di uguaglianza

In C #, ci sono due diversi tipi di uguaglianza: uguaglianza di riferimento e uguaglianza di valore. L'uguaglianza di valore è il significato comunemente inteso di uguaglianza: significa che due oggetti contengono gli stessi valori. Ad esempio, due numeri interi con valore 2 hanno l'uguaglianza di valore. L'uguaglianza di riferimento significa che non ci sono due oggetti da confrontare. Invece, ci sono due riferimenti a oggetti, entrambi riferiti allo stesso oggetto.

```
object a = new object();
object b = a;
System.Object.ReferenceEquals(a, b);  //returns true
```

Per i tipi di valore predefiniti, l'operatore di uguaglianza (==) restituisce true se i valori dei suoi operandi sono uguali, false altrimenti. Per i tipi di riferimento diversi da string, == restituisce true se i suoi due operandi si riferiscono allo stesso oggetto. Per il tipo di stringa, == confronta i valori delle stringhe.

```
// Numeric equality: True
Console.WriteLine((2 + 2) == 4);

// Reference equality: different objects,
// same boxed value: False.
object s = 1;
object t = 1;
Console.WriteLine(s == t);

// Define some strings:
string a = "hello";
string b = String.Copy(a);
string c = "hello";

// Compare string values of a constant and an instance: True
Console.WriteLine(a == b);

// Compare string references;
// a is a constant but b is an instance: False.
Console.WriteLine((object)a == (object)b);

// Compare string references, both constants
// have the same value, so string interning
// points to same reference: True.
Console.WriteLine((object)a == (object)c);
```

Leggi Operatore di uguaglianza online: https://riptutorial.com/it/csharp/topic/1491/operatore-di-uguaglianza

Capitolo 115: Operatore Null Coalescing

Sintassi

- var result = possibleNullObject ?? valore predefinito;

Parametri

Parametro	Dettagli
`possibleNullObject`	Il valore da verificare per il valore nullo. Se non è nullo, questo valore viene restituito. Deve essere un tipo nullable.
`defaultValue`	Il valore restituito se `possibleNullObject` è null. Deve essere lo stesso tipo `possibleNullObject` di `possibleNullObject` .

Osservazioni

Lo stesso operatore a coalescenza nulla è costituito da due caratteri consecutivi di punti interrogativi: `??`

È una scorciatoia per l'espressione condizionale:

```
possibleNullObject != null ? possibleNullObject : defaultValue
```

L'operando di sinistra (oggetto in fase di test) deve essere un tipo di valore o un tipo di riferimento nullable o si verificherà un errore di compilazione.

Il ?? l'operatore lavora sia per i tipi di riferimento che per i tipi di valore.

Examples

Utilizzo di base

L'utilizzo `null-coalescing operator (??)` consente di specificare un valore predefinito per un tipo nullable se l'operando di sinistra è `null` .

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString ?? "not provided"));
```

Live Demo su .NET Fiddle

Questo è logicamente equivalente a:

```
string testString = null;
if (testString == null)
{
    Console.WriteLine("The specified string is - not provided");
}
else
{
    Console.WriteLine("The specified string is - " + testString);
}
```

o usando l' operatore ternario (? :) operator:

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString == null ? "not provided" :
testString));
```

Fallimento nullo e concatenamento

L'operando di sinistra deve essere annullabile, mentre l'operando di destra può essere o meno. Il risultato verrà digitato di conseguenza.

Non annullabile

```
int? a = null;
int b = 3;
var output = a ?? b;
var type = output.GetType();

Console.WriteLine($"Output Type :{type}");
Console.WriteLine($"Output value :{output}");
```

Produzione:

> Tipo: System.Int32
> valore: 3

Visualizza la demo

nullable

```
int? a = null;
int? b = null;
var output = a ?? b;
```

`output` sarà di tipo `int?` e uguale a `b` , o `null` .

Coalescenza multipla

La coalescenza può anche essere eseguita in catene:

```
int? a = null;
int? b = null;
```

```
int c = 3;
var output = a ?? b ?? c;

var type = output.GetType();
Console.WriteLine($"Type :{type}");
Console.WriteLine($"value :{output}");
```

Produzione:

Tipo: System.Int32
valore: 3

Visualizza la demo

Concatenamento condizionale nullo

L'operatore null coalescing può essere utilizzato in tandem con l' operatore di propagazione null per fornire un accesso più sicuro alle proprietà degli oggetti.

```
object o = null;
var output = o?.ToString() ?? "Default Value";
```

Produzione:

Tipo: System.String
valore: valore predefinito

Visualizza la demo

Operatore coalescente Null con chiamate di metodo

L'operatore null coalescente semplifica l'assicurare che un metodo che può restituire `null` ricada su un valore predefinito.

Senza l'operatore null coalescente:

```
string name = GetName();

if (name == null)
    name = "Unknown!";
```

Con l'operatore null coalescente:

```
string name = GetName() ?? "Unknown!";
```

Usa esistente o crea nuovo

Uno scenario di utilizzo comune a cui questa funzione è davvero utile è quando si cerca un oggetto in una raccolta e occorre crearne uno nuovo se non esiste già.

```
IEnumerable<MyClass> myList = GetMyList();
var item = myList.SingleOrDefault(x => x.Id == 2) ?? new MyClass { Id = 2 };
```

Inizializzazione delle proprietà Lazy con operatore coalescente null

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars
{
    get { return _fooBars ?? (_fooBars = new List<FooBar>()); }
}
```

La prima volta che si accede alla proprietà `.FooBars` la variabile `_fooBars` verrà valutata come `null` , passando quindi all'istruzione di assegnazione assegna e valuterà il valore risultante.

Filo di sicurezza

Questo **non** è **un** modo **sicuro** per implementare le proprietà lazy. Per la pigrizia sicura dei thread, utilizzare la classe `Lazy<T>` integrata in .NET Framework.

Zucchero sintattico C # 6 usando corpi di espressione

Nota che dal C # 6, questa sintassi può essere semplificata usando il corpo di espressione per la proprietà:

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars => _fooBars ?? ( _fooBars = new List<FooBar>() );
```

Gli accessi successivi alla proprietà produrranno il valore memorizzato nella variabile `_fooBars` .

Esempio nel modello MVVM

Questo è spesso usato quando si implementano comandi nel pattern MVVM. Invece di inizializzare i comandi con entusiasmo con la costruzione di un viewmodel, i comandi vengono inizializzati pigramente usando questo modello come segue:

```
private ICommand _actionCommand = null;
public ICommand ActionCommand =>
   _actionCommand ?? ( _actionCommand = new DelegateCommand( DoAction ) );
```

Leggi Operatore Null Coalescing online: https://riptutorial.com/it/csharp/topic/37/operatore-null-coalescing

Capitolo 116: operatori

introduzione

In C #, un operatore è un elemento di programma che viene applicato a uno o più operandi in un'espressione o istruzione. Gli operatori che accettano un operando, come l'operatore di incremento (++) o nuovo, sono indicati come operatori unari. Gli operatori che accettano due operandi, come gli operatori aritmetici (+, -, *, /), sono indicati come operatori binari. Un operatore, l'operatore condizionale (? :), prende tre operandi ed è l'unico operatore ternario in C #.

Sintassi

- operatore statico pubblico OperandType operatoreSymbol (OperandType operando1)
- operatore statico pubblico OperandType operatoreSymbol (OperandType operand1, OperandType2 operando2)

Parametri

Parametro	Dettagli
operatorSymbol	Sovraccarico dell'operatore, ad es. +, -, /, *
OperandType	Il tipo che verrà restituito dall'operatore sovraccarico.
operand1	Il primo operando da utilizzare nell'esecuzione dell'operazione.
operando2	Il secondo operando da utilizzare nell'esecuzione dell'operazione, quando si eseguono operazioni binarie.
dichiarazioni	Codice opzionale necessario per eseguire l'operazione prima di restituire il risultato.

Osservazioni

Tutti gli operatori sono definiti come `static methods` e non sono `virtual` e non vengono ereditati.

Precedenza dell'operatore

Tutti gli operatori hanno una "precedenza" particolare a seconda del gruppo in cui ricade l'operatore (gli operatori dello stesso gruppo hanno la stessa priorità). Significa che alcuni operatori saranno applicati prima degli altri. Quello che segue è un elenco di gruppi (contenenti i rispettivi operatori) ordinati per precedenza (prima i più alti):

- **Operatori primari**

- `ab` - Accesso membri.
- `a?.b` - Accesso ai membri condizionali Null.
- `->` - Dereferencing del puntatore combinato con l'accesso dei membri.
- `f(x)` - Richiamo funzione.
- `a[x]` - Indicizzatore.
- `a?[x]` - Indicatore condizionale nullo.
- `x++` - Incremento postfisso.
- `x--` - `x--` Postfix.
- `new` - Digitare l'istanza.
- `default(T)` - Restituisce il valore inizializzato predefinito di tipo `T`
- `typeof` - Restituisce l'oggetto `Type` dell'operando.
- `checked` : abilita il controllo di overflow numerico.
- `unchecked` : disattiva il controllo di overflow numerico.
- `delegate` - Dichiara e restituisce un'istanza delegata.
- `sizeof` - Restituisce la dimensione in byte dell'operando di tipo.

- **Operatori unari**

  - `+x` - Restituisce `x` .
  - `-x` - Negazione numerica.
  - `!x` - Negazione logica.
  - `~x` - Complemento bit per bit / dichiara i distruttori.
  - `++x` - Incremento del prefisso.
  - `--x` - `--x` prefisso.
  - `(T)x` - Tipologia casting.
  - `await` - Aspetta un `Task` .
  - `&x` - Restituisce l'indirizzo (puntatore) di `x` .
  - `*x` - Dereferenziamento puntatore.

- **Operatori moltiplicativi**

  - `x * y` - Moltiplicazione.
  - `x / y` - Divisione.
  - `x % y` - Modulo.

- **Operatori additivi**

  - `x + y` - Aggiunta.
  - `x - y` - sottrazione.

- **Operatori di spostamento bit a bit**

  - `x << y` - Maiuscole a sinistra.
  - `x >> y` - Maiuscole a destra.

- **Operatori di test di tipo / relazionale**

  - `x < y` - Meno di.
  - `x > y` - Maggiore di.

- `x <= y` - Minore o uguale a.
- `x >= y` - Maggiore o uguale a.
- `is` - Digitare la compatibilità.
- `as` - Conversione di tipo.

- **Operatori di uguaglianza**
  - `x == y` - Uguaglianza.
  - `x != y` - Non uguale.
- **Logico E Operatore**
  - `x & y` - Logico / bit a bit AND.
- **Operatore logico XOR**
  - `x ^ y` - XOR logico / bit a bit.
- **Logico O Operatore**
  - `x | y` - OR logico / bit a bit.
- **Condizionale E Operatore**
  - `x && y` - Cortocircuito AND logico.
- **Operatore OR condizionale**
  - `x || y` - Cortocircuito logico OR.
- **Operatore a coalescenza nulla**
  - `x ?? y` - Restituisce `x` se non è nullo; altrimenti, restituisce `y` .
- **Operatore condizionale**
  - `x ? y : z` - Valuta / restituisce `y` se `x` è vero; altrimenti, valuta `z` .

Contenuto relativo

- Operatore Null Coalescing
- Operatore Null-Conditional
- nome dell'operatore

Examples

Operatori sovraccarichi

C # consente ai tipi definiti dall'utente di sovraccaricare gli operatori definendo funzioni membro statiche utilizzando la parola chiave `operator` .
L'esempio seguente illustra un'implementazione dell'operatore `+` .

Se abbiamo una classe `Complex` che rappresenta un numero complesso:

```
public struct Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }
}
```

E vogliamo aggiungere l'opzione per usare l'operatore `+` per questa classe. vale a dire:

```
Complex a = new Complex() { Real = 1, Imaginary = 2 };
Complex b = new Complex() { Real = 4, Imaginary = 8 };
Complex c = a + b;
```

Dovremo sovraccaricare l'operatore `+` per la classe. Questo viene fatto usando una funzione statica e la parola chiave `operator` :

```
public static Complex operator +(Complex c1, Complex c2)
{
   return new Complex
   {
       Real = c1.Real + c2.Real,
       Imaginary = c1.Imaginary + c2.Imaginary
   };
}
```

Operatori come `+` , `-` , `*` , `/` possono essere sovraccaricati. Ciò include anche operatori che non restituiscono lo stesso tipo (ad esempio, `==` e `!=` Possono essere sovraccaricati, nonostante il ritorno di booleani) Viene applicata anche la regola seguente relativa alle coppie.

Gli operatori di confronto devono essere sovraccaricati a coppie (ad esempio, se `<` è sovraccarico, `>` deve essere sovraccaricato).

Un elenco completo degli operatori sovraccaricabili (così come degli operatori non sovraccaricabili e delle restrizioni imposte ad alcuni operatori sovraccaricabili) può essere visto su MSDN - Operatori sovraccaricabili (C # Programming Guide) .

7.0

l'overloading `operator is` stato introdotto con il meccanismo di corrispondenza del modello di C # 7.0. Per i dettagli vedi Pattern Matching

Dato un tipo di `Cartesian` definito come segue

```
public class Cartesian
{
    public int X { get; }
    public int Y { get; }
```

```
}
```

Un `operator is` sovraccarico può essere definito ad esempio per coordinate `Polar`

```
public static class Polar
{
    public static bool operator is(Cartesian c, out double R, out double Theta)
    {
        R = Math.Sqrt(c.X*c.X + c.Y*c.Y);
        Theta = Math.Atan2(c.Y, c.X);
        return c.X != 0 || c.Y != 0;
    }
}
```

che può essere usato in questo modo

```
var c = Cartesian(3, 4);
if (c is Polar(var R, *))
{
    Console.WriteLine(R);
}
```

(L'esempio è tratto dalla documentazione di corrispondenza del modello Roslyn)

Operatori relazionali

È uguale a

Controlla se gli operandi forniti (argomenti) sono uguali

```
"a" == "b"     // Returns false.
"a" == "a"     // Returns true.
1 == 0         // Returns false.
1 == 1         // Returns true.
false == true  // Returns false.
false == false // Returns true.
```

A differenza di Java, l'operatore di confronto delle uguaglianze funziona in modo nativo con le stringhe.

L'operatore di confronto di uguaglianza funzionerà con operandi di tipi diversi se esiste un cast implicito da uno all'altro. Se non esiste alcun cast implicito adatto, è possibile chiamare un cast esplicito o utilizzare un metodo per convertire in un tipo compatibile.

```
1 == 1.0              // Returns true because there is an implicit cast from int to double.
new Object() == 1.0   // Will not compile.
MyStruct.AsInt() == 1 // Calls AsInt() on MyStruct and compares the resulting int with 1.
```

A differenza di Visual Basic.NET, l'operatore di confronto delle uguaglianze non è uguale all'operatore di assegnazione dell'uguaglianza.

```
var x = new Object();
var y = new Object();
x == y // Returns false, the operands (objects in this case) have different references.
x == x // Returns true, both operands have the same reference.
```

Da non confondere con l'operatore di assegnazione (`=`).

Per i tipi di valore, l'operatore restituisce `true` se entrambi gli operandi hanno valore uguale. Per i tipi di riferimento, l'operatore restituisce `true` se entrambi gli operandi sono uguali in *riferimento* (non valore). Un'eccezione è che gli oggetti stringa saranno confrontati con l'uguaglianza dei valori.

Non uguale

Controlla se gli operandi forniti *non* sono uguali.

```
"a" != "b"     // Returns true.
"a" != "a"     // Returns false.
1 != 0         // Returns true.
1 != 1         // Returns false.
false != true  // Returns true.
false != false // Returns false.

var x = new Object();
var y = new Object();
x != y // Returns true, the operands have different references.
x != x // Returns false, both operands have the same reference.
```

Questo operatore restituisce in modo efficace il risultato opposto a quello dell'operatore di uguale (`==`)

Più grande di

Controlla se il primo operando è maggiore del secondo operando.

```
3 > 5    //Returns false.
1 > 0    //Returns true.
2 > 2    //Return false.

var x = 10;
var y = 15;
x > y    //Returns false.
y > x    //Returns true.
```

Meno di

Controlla se il primo operando è inferiore al secondo operando.

```
2 < 4     //Returns true.
1 < -3    //Returns false.
2 < 2     //Return false.

var x = 12;
var y = 22;
```

```
x < y    //Returns true.
y < x    //Returns false.
```

Maggiore di uguale a

Controlla se il primo operando è maggiore di uguale al secondo operando.

```
7 >= 8    //Returns false.
0 >= 0    //Returns true.
```

Meno di uguale a

Controlla se il primo operando è inferiore al secondo operando.

```
2 <= 4    //Returns true.
1 <= -3    //Returns false.
1 <= 1     //Returns true.
```

Operatori di cortocircuito

Per definizione, gli operatori booleani di cortocircuito valuteranno solo il secondo operando se il primo operando non è in grado di determinare il risultato complessivo dell'espressione.

Significa che, se si utilizza && operator come *firstCondition e& secondCondition* , verrà valutata *secondCondition* solo quando *firstCondition* è true e ofcource il risultato complessivo sarà true solo se entrambi di *firstOperand* e *secondOperand* vengono valutati su true. Questo è utile in molti scenari, ad esempio immagina di voler controllare mentre il tuo elenco ha più di tre elementi ma devi anche controllare se l'elenco è stato inizializzato per non essere eseguito in *NullReferenceException* . Puoi ottenerlo come di seguito:

```
bool hasMoreThanThreeElements = myList != null && mList.Count > 3;
```

mList.Count> 3 non verrà controllato fino a myList! = null è soddisfatto.

AND logico

`&&` è la controparte in cortocircuito dell'operatore AND (`&`) standard booleano.

```
var x = true;
var y = false;

x && x // Returns true.
x && y // Returns false (y is evaluated).
y && x // Returns false (x is not evaluated).
y && y // Returns false (right y is not evaluated).
```

OR logico

`||` è la controparte in cortocircuito dell'operatore standard booleano OR (`|`).

```
var x = true;
var y = false;

x || x // Returns true (right x is not evaluated).
x || y // Returns true (y is not evaluated).
y || x // Returns true (x and y are evaluated).
y || y // Returns false (y and y are evaluated).
```

Esempio di utilizzo

```
if(object != null && object.Property)
// object.Property is never accessed if object is null, because of the short circuit.
    Action1();
else
    Action2();
```

taglia di

Restituisce un `int` che `int` la dimensione di un tipo * in byte.

```
sizeof(bool)    // Returns 1.
sizeof(byte)    // Returns 1.
sizeof(sbyte)   // Returns 1.
sizeof(char)    // Returns 2.
sizeof(short)   // Returns 2.
sizeof(ushort)  // Returns 2.
sizeof(int)     // Returns 4.
sizeof(uint)    // Returns 4.
sizeof(float)   // Returns 4.
sizeof(long)    // Returns 8.
sizeof(ulong)   // Returns 8.
sizeof(double)  // Returns 8.
sizeof(decimal) // Returns 16.
```

** Supporta solo alcuni tipi primitivi in un contesto sicuro.*

In un contesto non sicuro, `sizeof` può essere utilizzato per restituire la dimensione di altri tipi e strutture primitive.

```
public struct CustomType
{
    public int value;
}

static void Main()
{
    unsafe
    {
        Console.WriteLine(sizeof(CustomType)); // outputs: 4
    }
}
```

Sovraccarico di operatori di uguaglianza

Sovraccaricare solo gli operatori di uguaglianza non è sufficiente. In diverse circostanze, è possibile chiamare tutti i seguenti:

1. `object.Equals` e `object.GetHashCode`
2. `IEquatable<T>.Equals` (facoltativo, consente di evitare la boxe)
3. `operator ==` e `operator !=` (facoltativo, consente di utilizzare gli operatori)

Quando si `Equals` override di `Equals` , anche `GetHashCode` deve essere sostituito. Quando si implementa `Equals` , ci sono molti casi speciali: confronto con oggetti di un tipo diverso, confronto con il sé ecc.

Quando NON viene sovrascritto il metodo `Equals` e l'operatore `==` comportano diversamente per le classi e le strutture. Per le classi vengono confrontati solo i riferimenti e per i valori delle strutture le proprietà vengono confrontate tramite la riflessione, cosa può influire negativamente sulle prestazioni. `==` non può essere usato per confrontare le strutture a meno che non sia sovrascritto.

Generalmente l'operazione di uguaglianza deve rispettare le seguenti regole:

- Non deve *generare eccezioni* .
- Reflexivity: `A` sempre uguale a `A` (potrebbe non essere vero per i valori `NULL` in alcuni sistemi).
- Transitvity: se `A` uguale a `B` , e `B` uguale a `C` , allora `A` uguale a `C`
- Se `A` uguale a `B` , allora `A` e `B` hanno uguali codici hash.
- Indipendenza dell'albero di successione: se `B` e `C` sono istanze di `Class2` ereditate da `Class1` : `Class1.Equals(A,B)` deve sempre restituire lo stesso valore della chiamata a `Class2.Equals(A,B)` .

```
class Student : IEquatable<Student>
{
    public string Name { get; set; } = "";

    public bool Equals(Student other)
    {
        if (ReferenceEquals(other, null)) return false;
        if (ReferenceEquals(other, this)) return true;
        return string.Equals(Name, other.Name);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Name?.GetHashCode() ?? 0;
    }

    public static bool operator ==(Student left, Student right)
    {
        return Equals(left, right);
    }
```

```
    public static bool operator !=(Student left, Student right)
    {
        return !Equals(left, right);
    }
}
```

Operatori membri della classe: accesso membri

```
var now = DateTime.UtcNow;
//accesses member of a class.  In this case the UtcNow property.
```

Operatori membri della classe: accesso con membri condizionali nulli

```
var zipcode = myEmployee?.Address?.ZipCode;
//returns null if the left operand is null.
//the above is the equivalent of:
var zipcode = (string)null;
if (myEmployee != null && myEmployee.Address != null)
    zipcode = myEmployee.Address.ZipCode;
```

Operatori membri della classe: chiamata funzione

```
var age = GetAge(dateOfBirth);
//the above calls the function GetAge passing parameter dateOfBirth.
```

Operatori membri della classe: indicizzazione degli oggetti aggregati

```
var letters = "letters".ToCharArray();
char letter = letters[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example we take the second character from the array
//by calling letters[1]
//NB: Array Indexing starts at 0; i.e. the first letter would be given by letters[0].
```

Operatori membri della classe: indicizzazione condizionale nullo

```
var letters = null;
char? letter = letters?[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example  rather than throwing an error because letters is null
//letter is assigned the value null
```

"Esclusivo o" Operatore

L'operatore per un "esclusivo o" (per XOR breve) è: ^

Questo operatore restituisce true quando uno, ma solo uno, dei bool forniti è vero.

```
true ^ false   // Returns true
false ^ true   // Returns true
false ^ false  // Returns false
true ^ true    // Returns false
```

Operatori Bit-Shifting

Gli operatori di spostamento consentono ai programmatori di regolare un numero intero spostando tutti i suoi bit a sinistra o a destra. Il seguente diagramma mostra l'effetto di spostare un valore a sinistra di una cifra.

Tasto maiuscolo di sinistra

```
uint value = 15;              // 00001111

uint doubled = value << 1;    // Result = 00011110 = 30
uint shiftFour = value << 4;  // Result = 11110000 = 240
```

Destra-Shift

```
uint value = 240;             // 11110000

uint halved = value >> 1;     // Result = 01111000 = 120
uint shiftFour = value >> 4;  // Result = 00001111 = 15
```

Operatori di cast impliciti ed espliciti

C # consente ai tipi definiti dall'utente di controllare l'assegnazione e il cast attraverso l'uso di parole chiave `explicit` e `implicit` . La firma del metodo assume la forma:

```
public static <implicit/explicit> operator <ResultingType>(<SourceType> myType)
```

Il metodo non può accettare altri argomenti, né può essere un metodo di istanza. Può, tuttavia, accedere a qualsiasi membro privato di tipo in cui è definito.

Un esempio di un cast `implicit` ed `explicit` :

```
public class BinaryImage
{
    private bool[] _pixels;

    public static implicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator bool[](BinaryImage im)
    {
        return im._pixels;
    }
}
```

Consentire la seguente sintassi del cast:

```
var binaryImage = new BinaryImage();
ColorImage colorImage = binaryImage; // implicit cast, note the lack of type
bool[] pixels = (bool[])binaryImage; // explicit cast, defining the type
```

Gli operatori del cast possono lavorare in entrambe le direzioni, andando *dal* tuo tipo e andando *al* tuo tipo:

```
public class BinaryImage
{
    public static explicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator BinaryImage(ColorImage cm)
    {
        return new BinaryImage(cm);
    }
}
```

Infine, la parola chiave `as` , che può essere coinvolta nel cast all'interno di una gerarchia di tipi, **non** è valida in questa situazione. Anche dopo aver definito un cast `explicit` o `implicit` , non puoi fare:

```
ColorImage cm = myBinaryImage as ColorImage;
```

Genererà un errore di compilazione.

Operatori binari con assegnazione

C # ha diversi operatori che possono essere combinati con un segno `=` per valutare il risultato dell'operatore e quindi assegnare il risultato alla variabile originale.

Esempio:

```
x += y
```

equivale a

```
x = x + y
```

Operatori di assegnazione:

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `&=`

- `|=`
- `^=`
- `<<=`
- `>>=`

? : Operatore ternario

Restituisce uno dei due valori in base al valore di un'espressione booleana.

Sintassi:

```
condition ? expression_if_true : expression_if_false;
```

Esempio:

```
string name = "Frank";
Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");
```

L'operatore ternario ha una associazione destra che consente di utilizzare espressioni ternarie composte. Questo viene fatto aggiungendo ulteriori equazioni ternarie nella posizione vera o falsa di un'equazione ternaria genitore. Bisogna fare attenzione per garantire la leggibilità, ma questa può essere una utile stenografia in alcune circostanze.

In questo esempio, un'operazione ternaria composta valuta una funzione di `clamp` e restituisce il valore corrente se è compreso nell'intervallo, il valore `min` se è inferiore all'intervallo o il valore `max` se è superiore all'intervallo.

```
light.intensity = Clamp(light.intensity, minLight, maxLight);

public static float Clamp(float val, float min, float max)
{
    return (val < min) ? min : (val > max) ? max : val;
}
```

Anche gli operatori ternari possono essere annidati, come ad esempio:

```
a ? b ? "a is true, b is true" : "a is true, b is false" : "a is false"

// This is evaluated from left to right and can be more easily seen with parenthesis:

a ? (b ? x : y) : z

// Where the result is x if a && b, y if a && !b, and z if !a
```

Quando si scrivono dichiarazioni ternarie composte, è comune utilizzare parentesi o rientranza per migliorare la leggibilità.

I tipi di *expression_if_true* e *expression_if_false* devono essere identici o deve esserci una conversione implicita da uno all'altro.

```
condition ? 3 : "Not three"; // Doesn't compile because `int` and `string` lack an implicit
```

```
conversion.

condition ? 3.ToString() : "Not three"; // OK because both possible outputs are strings.

condition ? 3 : 3.5; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.

condition ? 3.5 : 3; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.
```

I requisiti di tipo e conversione si applicano anche alle tue classi.

```
public class Car
{}

public class SportsCar : Car
{}

public class SUV : Car
{}

condition ? new SportsCar() : new Car(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new Car() : new SportsCar(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new SportsCar() : new SUV(); // Doesn't compile because there is no implicit
conversion from `SportsCar` to SUV or `SUV` to `SportsCar`. The compiler is not smart enough
to realize that both of them have an implicit conversion to `Car`.

condition ? new SportsCar() as Car : new SUV() as Car; // OK because both expressions evaluate
to a reference of type `Car`. The ternary operator will return a reference of type `Car`.
```

tipo di

Ottiene l'oggetto `System.Type` per un tipo.

```
System.Type type = typeof(Point)        //System.Drawing.Point
System.Type type = typeof(IDisposable)  //System.IDisposable
System.Type type = typeof(Colors)       //System.Drawing.Color
System.Type type = typeof(List<>)       //System.Collections.Generic.List`1[T]
```

Per ottenere il tipo di runtime, utilizzare il metodo `GetType` per ottenere `System.Type` dell'istanza corrente.

L'operatore `typeof` accetta un nome di tipo come parametro, che viene specificato al momento della compilazione.

```
public class Animal {}
public class Dog : Animal {}

var animal = new Dog();

Assert.IsTrue(animal.GetType() == typeof(Animal)); // fail, animal is typeof(Dog)
```

```
Assert.IsTrue(animal.GetType() == typeof(Dog));    // pass, animal is typeof(Dog)
Assert.IsTrue(animal is Animal);                   // pass, animal implements Animal
```

Operatore predefinito

Tipo di valore (dove T: struct)

I tipi di dati primitivi predefiniti, come `char` , `int` e `float` , nonché i tipi definiti dall'utente dichiarati con `struct` o `enum` . Il loro valore predefinito è `new T()` :

```
default(int)            // 0
default(DateTime)       // 0001-01-01 12:00:00 AM
default(char)           // '\0' This is the "null character", not a zero or a line break.
default(Guid)           // 00000000-0000-0000-0000-000000000000
default(MyStruct)       // new MyStruct()

// Note: default of an enum is 0, and not the first *key* in that enum
// so it could potentially fail the Enum.IsDefined test
default(MyEnum)         // (MyEnum)0
```

Tipo di riferimento (dove T: classe)

Qualsiasi tipo di `class` , `interface` , matrice o delegato. Il loro valore predefinito è `null` :

```
default(object)         // null
default(string)         // null
default(MyClass)        // null
default(IDisposable)    // null
default(dynamic)        // null
```

nome dell'operatore

Restituisce una stringa che rappresenta il nome non qualificato di una `variable` , `type` o `member` .

```
int counter = 10;
nameof(counter); // Returns "counter"
Client client = new Client();
nameof(client.Address.PostalCode)); // Returns "PostalCode"
```

L'operatore `nameof` stato introdotto in C # 6.0. Viene valutato in fase di compilazione e il valore stringa restituito viene inserito inline dal compilatore, quindi può essere utilizzato nella maggior parte dei casi in cui è possibile utilizzare la stringa costante (ad esempio, le etichette `case` in un'istruzione `switch` , attributi, ecc. .). Può essere utile in casi come le eccezioni di sollevamento e registrazione, gli attributi, i link di azione MVC, ecc ...

?. (Operatore condizionale nullo)

6.0

Introdotto in C # 6.0 , il Null Conditional Operator `?.` immediatamente restituire `null` se l'espressione sul suo lato sinistro restituisce `null` , invece di lanciare un `NullReferenceException` . Se il suo lato sinistro valuta un valore non `null` , viene trattato come un normale `.` operatore. Si noti che poiché potrebbe restituire `null` , il suo tipo restituito è sempre un tipo nullable. Ciò significa che per una struttura o un tipo primitivo, è racchiuso in un `Nullable<T>` .

```
var bar = Foo.GetBar()?.Value; // will return null if GetBar() returns null
var baz = Foo.GetBar()?.IntegerValue; // baz will be of type Nullable<int>, i.e. int?
```

Questo è utile quando si attivano gli eventi. Normalmente dovresti racchiudere la chiamata dell'evento in un'istruzione if che verifica il `null` e innalza l'evento in seguito, che introduce la possibilità di una race condition. Utilizzando l'operatore condizionale Null, questo può essere risolto nel seguente modo:

```
event EventHandler<string> RaiseMe;
RaiseMe?.Invoke("Event raised");
```

Postfix e prefisso incremento e decremento

L'incremento Postfix `X++` aggiungerà `1` a `x`

```
var x = 42;
x++;
Console.WriteLine(x); // 43
```

Il decremento di Postfix `X--` ne sottrarrà uno

```
var x = 42
x--;
Console.WriteLine(x); // 41
```

`++x` è chiamato incremento prefisso incrementa il valore di x e quindi restituisce x mentre `x++` restituisce il valore di x e quindi incrementi

```
var x = 42;
Console.WriteLine(++x); // 43
System.out.println(x); // 43
```

mentre

```
var x = 42;
Console.WriteLine(x++); // 42
System.out.println(x); // 43
```

entrambi sono comunemente usati in loop

```
for(int i = 0; i < 10; i++)
{
}
```

=> Operatore Lambda

3.0

L'operatore `=>` ha la stessa precedenza dell'operatore di assegnazione `=` ed è associato a destra.

Viene utilizzato per dichiarare espressioni lambda e inoltre è ampiamente utilizzato con query LINQ :

```
string[] words = { "cherry", "apple", "blueberry" };

int shortestWordLength = words.Min((string w) => w.Length); //5
```

Se utilizzato nelle estensioni o nelle query LINQ, il tipo di oggetti può essere saltato in genere poiché viene dedotto dal compilatore:

```
int shortestWordLength = words.Min(w => w.Length); //also compiles with the same result
```

La forma generale dell'operatore lambda è la seguente:

```
(input parameters) => expression
```

I parametri dell'espressione lambda sono specificati prima `=>` operatore e l'espressione / istruzione / blocco effettivo da eseguire è a destra dell'operatore:

```
// expression
(int x, string s) => s.Length > x

// expression
(int x, int y) => x + y

// statement
(string x) => Console.WriteLine(x)

// block
(string x) => {
        x += " says Hello!";
        Console.WriteLine(x);
    }
```

Questo operatore può essere utilizzato per definire facilmente i delegati, senza scrivere un metodo esplicito:

```
delegate void TestDelegate(string s);

TestDelegate myDelegate = s => Console.WriteLine(s + " World");

myDelegate("Hello");
```

invece di

```
void MyMethod(string s)
{
    Console.WriteLine(s + " World");
}

delegate void TestDelegate(string s);

TestDelegate myDelegate = MyMethod;

myDelegate("Hello");
```

Operatore di assegnazione '='

L'operatore di assegnazione `=` imposta il valore dell'operando della mano sinistra al valore dell'operando di destra, e restituisce quel valore:

```
int a = 3;     // assigns value 3 to variable a
int b = a = 5; // first assigns value 5 to variable a, then does the same for variable b
Console.WriteLine(a = 3 + 4); // prints 7
```

?? Operatore Null Coalescing

L'operatore Null-Coalescing `??` restituirà il lato sinistro quando non è nullo. Se è nullo, restituirà il lato destro.

```
object foo = null;
object bar = new object();

var c = foo ?? bar;
//c will be bar since foo was null
```

Il `??` l'operatore può essere incatenato che consente la rimozione di `if` controlli.

```
//config will be the first non-null returned.
var config = RetrieveConfigOnMachine() ??
             RetrieveConfigFromService() ??
             new DefaultConfiguration();
```

Leggi operatori online: https://riptutorial.com/it/csharp/topic/18/operatori

Capitolo 117: Operatori non condizionali

Sintassi

- ? X .Y; // null se X è nullo XY
- ?? X .Y .Z; // null se X è null o Y è null else XYZ
- ? X [index]; // null se X è null else X [index]
- ? X .ValueMethod (); // null se X è null else il risultato di X.ValueMethod ();
- ? X .VoidMethod (); // non fare nulla se X è null altrimenti chiama X.VoidMethod ();

Osservazioni

Si noti che quando si utilizza l'operatore null coalescing su un tipo di valore `T` si ottiene un back `Nullable<T>` null `Nullable<T>` .

Examples

Operatore Null-Conditional

Il `?.` l'operatore è zucchero sintattico per evitare verosimili controlli null. È anche noto come operatore di navigazione sicura .

Classe utilizzata nell'esempio seguente:

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person Spouse { get; set; }
}
```

Se un oggetto è potenzialmente nullo (come una funzione che restituisce un tipo di riferimento), l'oggetto deve prima essere verificato per null per impedire una possibile `NullReferenceException` . Senza l'operatore condizionale nullo, questo apparirebbe:

```
Person person = GetPerson();

int? age = null;
if (person != null)
    age = person.Age;
```

Lo stesso esempio con l'operatore null-condizionale:

```
Person person = GetPerson();

var age = person?.Age;    // 'age' will be of type 'int?', even if 'person' is not null
```

Concatenare l'operatore

L'operatore null-condizionale può essere combinato sui membri e sotto-membri di un oggetto.

```
// Will be null if either `person` or `person.Spouse` are null
int? spouseAge = person?.Spouse?.Age;
```

Combinazione con l'operatore Null-Coalescing

L'operatore null-condizionale può essere combinato con l' operatore null-coalescing per fornire un valore predefinito:

```
// spouseDisplayName will be "N/A" if person, Spouse, or Name is null
var spouseDisplayName = person?.Spouse?.Name ?? "N/A";
```

L'indice Null-Conditional

Allo stesso modo del `?.` operatore, l'operatore indice condizionale null verifica i valori nulli durante l'indicizzazione in una raccolta che può essere nullo.

```
string item = collection?[index];
```

è zucchero sintattico per

```
string item = null;
if(collection != null)
{
    item = collection[index];
}
```

Evitare NullReferenceExceptions

```
var person = new Person
{
    Address = null;
};

var city = person.Address.City; //throws a NullReferenceException
var nullableCity = person.Address?.City; //returns the value of null
```

Questo effetto può essere concatenato:

```
var person = new Person
{
    Address = new Address
    {
        State = new State
        {
```

```
            Country = null
        }
    }
};

// this will always return a value of at least "null" to be stored instead
// of throwing a NullReferenceException
var countryName = person?.Address?.State?.Country?.Name;
```

L'operatore con condizioni null può essere utilizzato con il metodo di estensione

Il metodo di estensione può funzionare su riferimenti null , ma puoi usare `?.` per null controllare in ogni caso.

```
public class Person
{
    public string Name {get; set;}
}

public static class PersonExtensions
{
    public static int GetNameLength(this Person person)
    {
        return person == null ? -1 : person.Name.Length;
    }
}
```

Normalmente, il metodo verrà attivato per riferimenti `null` e restituirà -1:

```
Person person = null;
int nameLength = person.GetNameLength(); // returns -1
```

Usando `?.` il metodo non verrà attivato per riferimenti `null` e il tipo è `int?` :

```
Person person = null;
int? nameLength = person?.GetNameLength(); // nameLength is null.
```

Questo comportamento è in realtà previsto dal modo in cui il `?.` L'operatore lavora: eviterà di effettuare chiamate al metodo di istanza per istanze nulle, al fine di evitare `NullReferenceExceptions` . Tuttavia, la stessa logica si applica al metodo di estensione, nonostante la differenza su come viene dichiarato il metodo.

Per ulteriori informazioni sul motivo per cui viene chiamato il metodo di estensione nel primo esempio, consultare i metodi di estensione - documentazione di controllo nullo .

Leggi Operatori non condizionali online: https://riptutorial.com/it/csharp/topic/41/operatori-non-condizionali

Capitolo 118: Operazioni stringhe comuni

Examples

Divisione di una stringa in base a un carattere specifico

```
string helloWorld = "hello world, how is it going?";
string[] parts1 = helloWorld.Split(',');

//parts1: ["hello world", " how is it going?"]

string[] parts2 = helloWorld.Split(' ');

//parts2: ["hello", "world,", "how", "is", "it", "going?"]
```

Ottenere sottostringhe di una determinata stringa

```
string helloWorld = "Hello World!";
string world = helloWorld.Substring(6); //world = "World!"
string hello = helloWorld.Substring(0,5); // hello = "Hello"
```

`Substring` riporta la stringa in alto da un dato indice, o tra due indici (entrambi inclusi).

Determina se una stringa inizia con una determinata sequenza

```
string HelloWorld = "Hello World";
HelloWorld.StartsWith("Hello"); // true
HelloWorld.StartsWith("Foo"); // false
```

Trovare una stringa all'interno di una stringa

Usando `System.String.Contains` puoi scoprire se una stringa particolare esiste all'interno di una stringa. Il metodo restituisce un valore booleano, vero se la stringa esiste altrimenti false.

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the
substring
```

Ritaglio di caratteri indesiderati su inizio e / o fine di archi.

`String.Trim()`

```
string x = "   Hello World!    ";
string y = x.Trim(); // "Hello World!"

string q = "{(Hi!*";
string r = q.Trim( '(', '*', '{' ); // "Hi!"
```

`String.TrimStart()` e `String.TrimEnd()`

```
string q = "{(Hi*";
string r = q.TrimStart( '{' ); // "(Hi*"
string s = q.TrimEnd( '*' );   // "{(Hi"
```

Formattazione di una stringa

Utilizzare il metodo `String.Format()` per sostituire uno o più elementi nella stringa con la rappresentazione stringa di un oggetto specificato:

```
String.Format("Hello {0} Foo {1}", "World", "Bar") //Hello World Foo Bar
```

Unire una serie di stringhe in una nuova

```
var parts = new[] { "Foo", "Bar", "Fizz", "Buzz"};
var joined = string.Join(", ", parts);

//joined = "Foo, Bar, Fizz, Buzz"
```

Riempimento di una stringa a una lunghezza fissa

```
string s = "Foo";
string paddedLeft = s.PadLeft(5);        // paddedLeft = "  Foo" (pads with spaces by default)
string paddedRight = s.PadRight(6, '+'); // paddedRight = "Foo+++"
string noPadded = s.PadLeft(2);          // noPadded = "Foo" (original string is never
shortened)
```

Costruisci una stringa da Array

Il metodo `String.Join` ci aiuterà a costruire una stringa da array / elenco di caratteri o stringa. Questo metodo accetta due parametri. Il primo è il delimitatore o il separatore che ti aiuterà a separare ogni elemento dell'array. E il secondo parametro è la matrice stessa.

Stringa dal `char array` :

```
string delimiter=",";
char[] charArray = new[] { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charArray);
```

Uscita : `a,b,c` se cambiamo il `delimiter` come `""` allora l'uscita diventerà `abc` .

Stringa da `List of char` :

```
string delimiter = "|";
List<char> charList = new List<char>() { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charList);
```

Uscita : `a|b|c`

Stringa dall'elenco `List of Strings` :

```
string delimiter = " ";
List<string> stringList = new List<string>() { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringList);
```

Uscita : `Ram is a boy`

Stringa da `array of strings` :

```
string delimiter = "_";
string[] stringArray = new [] { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringArray);
```

Uscita : `Ram_is_a_boy`

Formattazione usando ToString

Di solito stiamo usando il metodo `String.Format` per scopi di formattazione, il `.ToString` viene solitamente utilizzato per convertire altri tipi in stringa. Possiamo specificare il formato insieme al metodo ToString mentre la conversione è in corso, quindi possiamo evitare una formattazione aggiuntiva. Lasciami spiegare come funziona con diversi tipi;

Intero alla stringa formattata:

```
int intValue = 10;
string zeroPaddedInteger = intValue.ToString("000"); // Output will be "010"
string customFormat = intValue.ToString("Input value is 0"); // output will be   "Input value
is 10"
```

da doppio a stringa formattata:

```
double doubleValue = 10.456;
string roundedDouble = doubleValue.ToString("0.00"); // output 10.46
string integerPart = doubleValue.ToString("00");    // output 10
string customFormat = doubleValue.ToString("Input value is 0.0");  // Input value is 10.5
```

Formattazione di DateTime usando ToString

```
DateTime currentDate = DateTime.Now; //  {7/21/2016 7:23:15 PM}
string dateTimeString = currentDate.ToString("dd-MM-yyyy HH:mm:ss"); // "21-07-2016 19:23:15"
string dateOnlyString = currentDate.ToString("dd-MM-yyyy"); // "21-07-2016"
string dateWithMonthInWords = currentDate.ToString("dd-MMMM-yyyy HH:mm:ss"); // "21-July-2016
19:23:15"
```

Ottenere caratteri x dal lato destro di una stringa

Visual Basic dispone di funzioni Left, Right e Mid che restituiscono caratteri da sinistra, destra e

centrale di una stringa. Questi metodi non esistono in C #, ma possono essere implementati con `Substring()` . Possono essere implementati come metodi di estensione come il seguente:

```
public static class StringExtensions
{
   /// <summary>
   /// VB Left function
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="numchars"></param>
   /// <returns>Left-most numchars characters</returns>
   public static string Left( this string stringparam, int numchars )
   {
      // Handle possible Null or numeric stringparam being passed
      stringparam += string.Empty;

      // Handle possible negative numchars being passed
      numchars = Math.Abs( numchars );

      // Validate numchars parameter
      if (numchars > stringparam.Length)
         numchars = stringparam.Length;

      return stringparam.Substring( 0, numchars );
   }

   /// <summary>
   /// VB Right function
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="numchars"></param>
   /// <returns>Right-most numchars characters</returns>
   public static string Right( this string stringparam, int numchars )
   {
      // Handle possible Null or numeric stringparam being passed
      stringparam += string.Empty;

      // Handle possible negative numchars being passed
      numchars = Math.Abs( numchars );

      // Validate numchars parameter
      if (numchars > stringparam.Length)
         numchars = stringparam.Length;

      return stringparam.Substring( stringparam.Length - numchars );
   }

   /// <summary>
   /// VB Mid function - to end of string
   /// </summary>
   /// <param name="stringparam"></param>
   /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
   /// <returns>Balance of string beginning at startindex character</returns>
   public static string Mid( this string stringparam, int startindex )
   {
      // Handle possible Null or numeric stringparam being passed
      stringparam += string.Empty;

      // Handle possible negative startindex being passed
      startindex = Math.Abs( startindex );
```

```
         // Validate numchars parameter
         if (startindex > stringparam.Length)
            startindex = stringparam.Length;

         // C# strings are zero-based, convert passed startindex
         return stringparam.Substring( startindex - 1 );
      }

      /// <summary>
      /// VB Mid function - for number of characters
      /// </summary>
      /// <param name="stringparam"></param>
      /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
      /// <param name="numchars">number of characters to return</param>
      /// <returns>Balance of string beginning at startindex character</returns>
      public static string Mid( this string stringparam, int startindex, int numchars)
      {
         // Handle possible Null or numeric stringparam being passed
         stringparam += string.Empty;

         // Handle possible negative startindex being passed
         startindex = Math.Abs( startindex );

         // Handle possible negative numchars being passed
         numchars = Math.Abs( numchars );

         // Validate numchars parameter
         if (startindex > stringparam.Length)
            startindex = stringparam.Length;

         // C# strings are zero-based, convert passed startindex
         return stringparam.Substring( startindex - 1, numchars );

       }
    }
```

Questo metodo di estensione può essere utilizzato come segue:

```
string myLongString = "Hello World!";
string myShortString = myLongString.Right(6);  // "World!"
string myLeftString = myLongString.Left(5);    // "Hello"
string myMidString1 = myLongString.Left(4);    // "lo World"
string myMidString2 = myLongString.Left(2,3);    // "ell"
```

Verifica della stringa vuota utilizzando String.IsNullOrEmpty () e String.IsNullOrWhiteSpace ()

```
string nullString = null;
string emptyString = "";
string whitespaceString = "    ";
string tabString = "\t";
string newlineString = "\n";
string nonEmptyString = "abc123";

bool result;
```

```
result = String.IsNullOrEmpty(nullString);            // true
result = String.IsNullOrEmpty(emptyString);           // true
result = String.IsNullOrEmpty(whitespaceString);      // false
result = String.IsNullOrEmpty(tabString);             // false
result = String.IsNullOrEmpty(newlineString);         // false
result = String.IsNullOrEmpty(nonEmptyString);        // false

result = String.IsNullOrWhiteSpace(nullString);       // true
result = String.IsNullOrWhiteSpace(emptyString);      // true
result = String.IsNullOrWhiteSpace(tabString);        // true
result = String.IsNullOrWhiteSpace(newlineString);    // true
result = String.IsNullOrWhiteSpace(whitespaceString); // true
result = String.IsNullOrWhiteSpace(nonEmptyString);   // false
```

Ottenere un carattere a un indice specifico ed enumerare la stringa

È possibile utilizzare il metodo `Substring` per ottenere qualsiasi numero di caratteri da una stringa in qualsiasi posizione specifica. Tuttavia, se vuoi solo un singolo carattere, puoi usare l'indicizzatore di stringhe per ottenere un singolo carattere in un dato indice come fai con un array:

```
string s = "hello";
char c = s[1]; //Returns 'e'
```

Si noti che il tipo restituito è `char` , a differenza del metodo `Substring` che restituisce un tipo di `string` .

Puoi anche usare l'indicizzatore per scorrere i caratteri della stringa:

```
string s = "hello";
foreach (char c in s)
    Console.WriteLine(c);
/********* This will print each character on a new line:
h
e
l
l
o
**********/
```

Converti numero decimale in formato binario, ottale ed esadecimale

1. Per convertire il numero decimale in formato binario utilizzare la **base 2**

   ```
   Int32 Number = 15;
   Console.WriteLine(Convert.ToString(Number, 2));  //OUTPUT : 1111
   ```

2. Per convertire il numero decimale in formato ottale, usa la **base 8**

   ```
   int Number = 15;
   Console.WriteLine(Convert.ToString(Number, 8));  //OUTPUT : 17
   ```

3. Per convertire il numero decimale in formato esadecimale utilizzare la **base 16**

```
var Number = 15;
Console.WriteLine(Convert.ToString(Number, 16));  //OUTPUT : f
```

Divisione di una stringa con un'altra stringa

```
string str = "this--is--a--complete--sentence";
string[] tokens = str.Split(new[] { "--" }, StringSplitOptions.None);
```

Risultato:

> ["questo", "è", "un", "completo", "frase"]

Invertire correttamente una stringa

La maggior parte delle volte in cui le persone devono invertire una stringa, lo fanno più o meno così:

```
char[] a = s.ToCharArray();
System.Array.Reverse(a);
string r = new string(a);
```

Tuttavia, ciò di cui queste persone non si rendono conto è che questo è in realtà sbagliato.
E non intendo per il controllo NULL mancante.

In realtà è sbagliato perché un Glyph / GraphemeCluster può consistere in diversi codepoint (ovvero personaggi).

Per capire perché è così, dobbiamo prima essere consapevoli del fatto che cosa significhi in realtà il termine "personaggio".

Riferimento:

> Il carattere è un termine sovraccarico che può significare molte cose.
>
> Un punto di codice è l'unità atomica di informazioni. Il testo è una sequenza di punti di codice. Ogni punto di codice è un numero a cui viene assegnato il significato dallo standard Unicode.
>
> Un grafema è una sequenza di uno o più punti di codice che vengono visualizzati come una singola unità grafica che un lettore riconosce come un singolo elemento del sistema di scrittura. Ad esempio, sia a che ä sono grafemi, ma possono essere costituiti da più punti di codice (ad esempio ä possono essere due punti di codice, uno per il carattere base uno seguito da uno per la diaresi, ma c'è anche un codice alternativo, legacy, singolo punto che rappresenta questo grafema). Alcuni punti di codice non fanno mai parte di alcun grafema (es. Il non-falegname a larghezza zero o gli override direzionali).
>
> Un glifo è un'immagine, solitamente memorizzata in un font (che è una raccolta di glifi), usata per rappresentare grafemi o parti di essi. I caratteri possono comporre più glifi in

una singola rappresentazione, ad esempio, se quanto sopra ä è un singolo punto di codice, un font può scegliere di renderlo come due glifi separati, spazialmente sovrapposti. Per OTF, le tabelle GSUB e GPOS del font contengono informazioni di sostituzione e posizionamento per far funzionare questo. Un font può contenere più glifi alternativi per lo stesso grafema.

Quindi in C #, un personaggio è in realtà un CodePoint.

Il che significa, se si inverte una stringa valida come `Les Misé rables` , che può assomigliare a questo

```
string s = "Les Mise\u0301rables";
```

come sequenza di caratteri, otterrai:

selbaêsMe seL

Come puoi vedere, l'accento è sul carattere R, invece del carattere e.
Sebbene string.reverse.reverse restituisca la stringa originale se entrambe le volte invertite il char array, questo tipo di inversione NON è sicuramente l'opposto della stringa originale.

Dovrai invertire solo ciascun GraphemeCluster.
Quindi, se fatto correttamente, si inverte una stringa come questa:

```
    private static System.Collections.Generic.List<string> GraphemeClusters(string s)
    {
        System.Collections.Generic.List<string> ls = new
System.Collections.Generic.List<string>();

        System.Globalization.TextElementEnumerator enumerator =
System.Globalization.StringInfo.GetTextElementEnumerator(s);
        while (enumerator.MoveNext())
        {
            ls.Add((string)enumerator.Current);
        }

        return ls;
    }


    // this
    private static string ReverseGraphemeClusters(string s)
    {
        if(string.IsNullOrEmpty(s) || s.Length == 1)
             return s;

        System.Collections.Generic.List<string> ls = GraphemeClusters(s);
        ls.Reverse();

        return string.Join("", ls.ToArray());
    }

    public static void TestMe()
    {
        string s = "Les Mise\u0301rables";
```

```
        // s = "noël";
        string r = ReverseGraphemeClusters(s);

        // This would be wrong:
        // char[] a = s.ToCharArray();
        // System.Array.Reverse(a);
        // string r = new string(a);

        System.Console.WriteLine(r);
    }
```

E - oh gioia - ti renderai conto che se lo fai correttamente, funzionerà anche per le lingue asiatico / sud-asiatico / est asiatico (e francese / svedese / norvegese, ecc.) ...

Sostituzione di una stringa all'interno di una stringa

Utilizzando il metodo `System.String.Replace` , è possibile sostituire parte di una stringa con un'altra stringa.

```
string s = "Hello World";
 s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

Tutte le occorrenze della stringa di ricerca vengono sostituite.

Questo metodo può anche essere utilizzato per rimuovere parte di una stringa, utilizzando il campo `String.Empty` :

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Modifica del caso di caratteri all'interno di una stringa

La classe `System.String` supporta un numero di metodi per la conversione tra caratteri maiuscoli e minuscoli in una stringa.

- `System.String.ToLowerInvariant` viene utilizzato per restituire un oggetto String convertito in lettere minuscole.
- `System.String.ToUpperInvariant` viene utilizzato per restituire un oggetto String convertito in maiuscolo.

Nota: la ragione per utilizzare le versioni *invarianti* di questi metodi è impedire la produzione di lettere impreviste specifiche della cultura. Questo è spiegato qui in dettaglio .

Esempio:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Si noti che è *possibile* scegliere di specificare una **cultura** specifica durante la conversione in

lettere minuscole e maiuscole utilizzando i metodi String.ToLower (CultureInfo) e String.ToUpper (CultureInfo) di conseguenza.

Concatena una serie di stringhe in una singola stringa

Il metodo `System.String.Join` consente di concatenare tutti gli elementi in una matrice di stringhe, utilizzando un separatore specificato tra ciascun elemento:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

Concatenazione di stringhe

String La concatenazione può essere eseguita utilizzando il metodo `System.String.Concat` o (molto più semplice) utilizzando l'operatore `+` :

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

In C # 6 questo può essere fatto come segue:

```
string concat = $"{first},{second}";
```

Leggi Operazioni stringhe comuni online: https://riptutorial.com/it/csharp/topic/73/operazioni-stringhe-comuni

Capitolo 119: Parola chiave rendimento

introduzione

Quando si utilizza la parola chiave yield in un'istruzione, si indica che il metodo, l'operatore o l'opzione di accesso in cui appare è un iteratore. L'utilizzo di yield per definire un iteratore rimuove la necessità di una classe extra esplicita (la classe che contiene lo stato per un'enumerazione) quando si implementa il modello IEnumerable e IEnumerator per un tipo di raccolta personalizzato.

Sintassi

- yield return [TYPE]
- cedimento

Osservazioni

Inserendo la parola chiave `yield` in un metodo con il tipo restituito di `IEnumerable` , `IEnumerable<T>` , `IEnumerator` o `IEnumerator<T>` indica al compilatore di generare un'implementazione del tipo restituito (`IEnumerable` o `IEnumerator`) che, una volta eseguito il loop, esegue il metodo fino a ciascun "rendimento" per ottenere ogni risultato.

La parola chiave `yield` è utile quando si desidera restituire "il prossimo" elemento di una sequenza teoricamente illimitata, quindi calcolare l'intera sequenza in anticipo sarebbe impossibile, o quando calcolare la sequenza completa di valori prima di tornare comporterebbe una pausa indesiderata per l'utente .

`yield break` può essere utilizzata anche per interrompere la sequenza in qualsiasi momento.

Poiché la parola chiave `yield` richiede un tipo di interfaccia iteratore come tipo restituito, ad esempio `IEnumerable<T>` , non è possibile utilizzarlo in un metodo async poiché restituisce un oggetto `Task<IEnumerable<T>>` .

Ulteriori letture

- https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx

Examples

Uso semplice

La parola chiave `yield` viene utilizzata per definire una funzione che restituisce un oggetto `IEnumerable` o `IEnumerator` (nonché le varianti generiche derivate) i cui valori vengono generati pigramente mentre un chiamante esegue iterazioni sulla raccolta restituita. Maggiori informazioni sullo scopo nella sezione commenti .

L'esempio seguente ha un'istruzione return return all'interno di un ciclo `for` .

```
public static IEnumerable<int> Count(int start, int count)
{
    for (int i = 0; i <= count; i++)
    {
        yield return start + i;
    }
}
```

Quindi puoi chiamarlo:

```
foreach (int value in Count(start: 4, count: 10))
{
    Console.WriteLine(value);
}
```

Uscita console

```
4
5
6
...
14
```

Live Demo su .NET Fiddle

Ogni iterazione del corpo `foreach` crea una chiamata alla funzione `Count` iteratore. Ogni chiamata alla funzione iteratore procede alla successiva esecuzione `yield return` , che si verifica durante la successiva iterazione del ciclo `for` .

Più l'uso pertinente

```
public IEnumerable<User> SelectUsers()
{
    // Execute an SQL query on a database.
    using (IDataReader reader = this.Database.ExecuteReader(CommandType.Text, "SELECT Id, Name
FROM Users"))
    {
        while (reader.Read())
        {
            int id = reader.GetInt32(0);
            string name = reader.GetString(1);
            yield return new User(id, name);
        }
    }
}
```

Ci sono altri modi per ottenere un `IEnumerable<User>` da un database SQL, questo dimostra semplicemente che puoi usare `yield` per trasformare qualsiasi cosa abbia una semantica di "sequence of elements" in un `IEnumerable<T>` che qualcuno può iterare su .

Risoluzione anticipata

È possibile estendere la funzionalità dei metodi di `yield` esistenti passando uno o più valori o elementi che potrebbero definire una condizione di terminazione all'interno della funzione chiamando un'interruzione di `yield break` per interrompere l'esecuzione del ciclo interno.

```
public static IEnumerable<int> CountUntilAny(int start, HashSet<int> earlyTerminationSet)
{
    int curr = start;

    while (true)
    {
        if (earlyTerminationSet.Contains(curr))
        {
            // we've hit one of the ending values
            yield break;
        }

        yield return curr;

        if (curr == Int32.MaxValue)
        {
            // don't overflow if we get all the way to the end; just stop
            yield break;
        }

        curr++;
    }
}
```

Il metodo sopra riportato itererà da una data posizione di `start` fino a `earlyTerminationSet` si incontra uno dei valori all'interno del primo `earlyTerminationSet` .

```
// Iterate from a starting point until you encounter any elements defined as
// terminating elements
var terminatingElements = new HashSet<int>{ 7, 9, 11 };
// This will iterate from 1 until one of the terminating elements is encountered (7)
foreach(var x in CountUntilAny(1,terminatingElements))
{
    // This will write out the results from 1 until 7 (which will trigger terminating)
    Console.WriteLine(x);
}
```

Produzione:

1
2
3
4
5
6

Live Demo su .NET Fiddle

Controllo corretto degli argomenti

Un metodo iteratore non viene eseguito fino a quando il valore di ritorno non viene enumerato. È quindi vantaggioso affermare le precondizioni al di fuori dell'iteratore.

```
public static IEnumerable<int> Count(int start, int count)
{
    // The exception will throw when the method is called, not when the result is iterated
    if (count < 0)
        throw new ArgumentOutOfRangeException(nameof(count));

    return CountCore(start, count);
}

private static IEnumerable<int> CountCore(int start, int count)
{
    // If the exception was thrown here it would be raised during the first MoveNext()
    // call on the IEnumerator, potentially at a point in the code far away from where
    // an incorrect value was passed.
    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}
```

Calling Side Code (Uso):

```
// Get the count
var count = Count(1,10);
// Iterate the results
foreach(var x in count)
{
    Console.WriteLine(x);
}
```

Produzione:

1
2
3
4
5
6
7
8
9
10

Live Demo su .NET Fiddle

Quando un metodo usa `yield` per generare un enumerabile, il compilatore crea una macchina a stati che, una volta iterata, eseguirà il codice fino a un `yield` . Quindi restituisce l'articolo restituito e salva il suo stato.

Ciò significa che non si scopriranno argomenti non validi (passaggio `null` ecc.) Quando si chiama

il metodo per la prima volta (poiché ciò crea la macchina di stato), solo quando si tenta di accedere al primo elemento (perché solo allora il codice all'interno del il metodo viene eseguito dalla macchina di stato). Racchiudendolo in un metodo normale che prima controlla gli argomenti è possibile controllarli quando viene chiamato il metodo. Questo è un esempio di veloce fallimento.

Quando si utilizza C # 7+, la funzione `CountCore` può essere opportunamente nascosta nella funzione `Count` come *funzione locale* . Vedi l'esempio qui .

Restituisce un altro Enumerable all'interno di un metodo che restituisce Enumerable

```
public IEnumerable<int> F1()
{
    for (int i = 0; i < 3; i++)
        yield return i;

    //return F2(); // Compile Error!!
    foreach (var element in F2())
        yield return element;
}

public int[] F2()
{
    return new[] { 3, 4, 5 };
}
```

Valutazione pigra

Solo quando l'istruzione `foreach` sposta sull'elemento successivo, il blocco iteratore valuta fino alla successiva dichiarazione di `yield` .

Considera il seguente esempio:

```
private IEnumerable<int> Integers()
{
    var i = 0;
    while(true)
    {
        Console.WriteLine("Inside iterator: " + i);
        yield return i;
        i++;
    }
}

private void PrintNumbers()
{
    var numbers = Integers().Take(3);
    Console.WriteLine("Starting iteration");

    foreach(var number in numbers)
    {
        Console.WriteLine("Inside foreach: " + number);
    }
```

```
}
```

Questo produrrà:

Inizia l'iterazione
All'interno dell'iteratore: 0
Dentro foreach: 0
All'interno di iteratore: 1
Dentro foreach: 1
All'interno di iteratore: 2
Dentro foreach: 2

Visualizza la demo

Come conseguenza:

- "Inizializzazione iterazione" viene stampata prima anche se il metodo iteratore è stato chiamato prima della riga che lo stampa perché la riga `Integers().Take(3);` non avvia effettivamente l'iterazione (non è stata effettuata alcuna chiamata a `IEnumerator.MoveNext()`)
- Le linee che stampano su console si alternano tra quella all'interno del metodo iteratore e quella all'interno del `foreach` , piuttosto che tutte quelle che si trovano nel metodo iteratore valutando prima
- Questo programma termina a causa del metodo `.Take()` , anche se il metodo iteratore ha un `while true` che non si interrompe mai.

Prova ... finalmente

Se un metodo iteratore ha un rendimento all'interno di una `try...finally` , l' `IEnumerator` restituito eseguirà l'istruzione `finally` quando viene chiamato `Dispose` , a condizione che il punto di valutazione corrente si trovi all'interno del blocco `try` .

Data la funzione:

```
private IEnumerable<int> Numbers()
{
    yield return 1;
    try
    {
        yield return 2;
        yield return 3;
    }
    finally
    {
        Console.WriteLine("Finally executed");
    }
}
```

Quando si chiama:

```
private void DisposeOutsideTry()
{
```

```
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

Quindi stampa:

    1

Visualizza la demo

Quando si chiama:

```
private void DisposeInsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

Quindi stampa:

    1
    2
    Finalmente giustiziato

Visualizza la demo

Usare yield per creare un IEnumerator quando si implementa IEnumerable

L'interfaccia `IEnumerable<T>` ha un unico metodo, `GetEnumerator()` , che restituisce un `IEnumerator<T>` .

Mentre la parola chiave `yield` può essere utilizzata per creare direttamente un `IEnumerable<T>` , può *anche* essere utilizzata esattamente nello stesso modo per creare un `IEnumerator<T>` . L'unica cosa che cambia è il tipo di ritorno del metodo.

Questo può essere utile se vogliamo creare la nostra classe che implementa `IEnumerable<T>` :

```
public class PrintingEnumerable<T> : IEnumerable<T>
{
    private IEnumerable<T> _wrapped;

    public PrintingEnumerable(IEnumerable<T> wrapped)
    {
        _wrapped = wrapped;
    }
```

```
    // This method returns an IEnumerator<T>, rather than an IEnumerable<T>
    // But the yield syntax and usage is identical.
    public IEnumerator<T> GetEnumerator()
    {
        foreach(var item in _wrapped)
        {
            Console.WriteLine("Yielding: " + item);
            yield return item;
        }
    }

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

(Si noti che questo particolare esempio è solo illustrativo e potrebbe essere implementato in modo più pulito con un singolo metodo iteratore che restituisce un oggetto `IEnumerable<T>` .)

Stimolante valutazione

La parola chiave `yield` consente una valutazione lazy della collezione. Il caricamento forzato dell'intera collezione in memoria si chiama **valutazione entusiasta** .

Il seguente codice mostra questo:

```
IEnumerable<int> myMethod()
{
    for(int i=0; i <= 8675309; i++)
    {
        yield return i;
    }
}
...
// define the iterator
var it = myMethod.Take(3);
// force its immediate evaluation
// list will contain 0, 1, 2
var list = it.ToList();
```

Calling `ToList` , `ToDictionary` o `ToArray` costringono la valutazione immediata dell'enumerazione, recuperando tutti gli elementi in una raccolta.

Esempio di valutazione pigro: numeri di Fibonacci

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics; // also add reference to System.Numberics

namespace ConsoleApplication33
{
    class Program
```

```
    {
        private static IEnumerable<BigInteger> Fibonacci()
        {
            BigInteger prev = 0;
            BigInteger current = 1;
            while (true)
            {
                yield return current;
                var next = prev + current;
                prev = current;
                current = next;
            }
        }

        static void Main()
        {
            // print Fibonacci numbers from 10001 to 10010
            var numbers = Fibonacci().Skip(10000).Take(10).ToArray();
            Console.WriteLine(string.Join(Environment.NewLine, numbers));
        }
    }
}
```

Come funziona sotto il cofano (consiglio di decompilare il file .exe risultante nello strumento IL Disaambler):

1. Il compilatore C # genera una classe che implementa `IEnumerable<BigInteger>` e `IEnumerator<BigInteger>` (`<Fibonacci>d__0` in ildasm).
2. Questa classe implementa una macchina a stati. Lo stato consiste nella posizione corrente nel metodo e nei valori delle variabili locali.
3. Il codice più interessante si trova nel metodo `bool IEnumerator.MoveNext()` . Fondamentalmente, cosa `MoveNext()` fare:
    - Ripristina lo stato attuale. Variabili come `prev` e `current` diventano campi nella nostra classe (`<current>5__2` e `<prev>5__1` in ildasm). Nel nostro metodo abbiamo due posizioni (`<>1__state`): prima alla parentesi graffa di apertura, seconda alla `yield return` .
    - Esegue il codice fino al prossimo `yield return` o `yield break` / `}` .
    - Per il valore `yield return` valore risultante viene salvato, quindi la proprietà `Current` può restituirlo. `true` viene restituito. A questo punto, lo stato corrente viene nuovamente salvato per la successiva `MoveNext` .
    - Per il metodo `yield break` / `}` restituisce solo il significato `false` che l'iterazione è stata eseguita.

Si noti inoltre che il 10001 ° numero è lungo 468 byte. La macchina a stati salva solo `current` variabili `current` e `prev` come campi. Mentre se vorremmo salvare tutti i numeri nella sequenza dal primo al 10000, la dimensione della memoria consumata sarà di oltre 4 megabyte. Quindi la valutazione pigra, se usata correttamente, può ridurre l'impronta della memoria in alcuni casi.

La differenza tra pausa e pausa di rendimento

L'uso della `yield break` rispetto alla `break` potrebbe non essere così ovvio come si potrebbe pensare. Ci sono molti cattivi esempi su Internet in cui l'uso dei due è intercambiabile e in realtà

non dimostra la differenza.

La parte confusa è che entrambe le parole chiave (o frasi chiave) hanno senso solo all'interno di cicli (`foreach` , `while` ...) Quindi quando scegliere l'una rispetto all'altra?

È importante rendersi conto che una volta che si utilizza la parola chiave `yield` in un metodo, si trasforma effettivamente il metodo in un iteratore . L'unico scopo di tale metodo è quindi di iterare su una collezione finita o infinita e produrre (produrre) i suoi elementi. Una volta che lo scopo è soddisfatto, non c'è motivo di continuare l'esecuzione del metodo. A volte, accade naturalmente con l'ultima parentesi di chiusura del metodo `}` . Ma a volte, vuoi terminare prematuramente il metodo. In un normale metodo (non iterativo) useresti la parola chiave `return` . Ma non puoi usare `return` in un iteratore, devi usare `yield break` . In altre parole, l' `yield break` per un iteratore equivale al `return` di un metodo standard. Mentre l'istruzione `break` termina semplicemente il ciclo più vicino.

Vediamo alcuni esempi:

```
/// <summary>
/// Yields numbers from 0 to 9
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9}</returns>
public static IEnumerable<int> YieldBreak()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Indicates that the iteration has ended, everything
            // from this line on will be ignored
            yield break;
        }
    }
    yield return 10; // This will never get executed
}
```

```
/// <summary>
/// Yields numbers from 0 to 10
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9,10}</returns>
public static IEnumerable<int> Break()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Terminates just the loop
```

```
                break;
            }
        }
        // Execution continues
        yield return 10;
    }
```

Leggi Parola chiave rendimento online: https://riptutorial.com/it/csharp/topic/61/parola-chiave-rendimento

Capitolo 120: parole

introduzione

Le parole chiave sono identificatori predefiniti e riservati con un significato speciale per il compilatore. Non possono essere usati come identificatori nel tuo programma senza il prefisso `@` . Ad esempio, `@if` è un identificatore legale ma non la parola chiave `if` .

Osservazioni

C # ha una collezione predefinita di "parole chiave" (o parole riservate) che hanno ciascuna una funzione speciale. Queste parole non possono essere utilizzate come identificatori (nomi per variabili, metodi, classi, ecc.) A meno che non siano preceduti da `@` .

- `abstract`
- `as`
- `base`
- `bool`
- `break`
- `byte`
- `case`
- `catch`
- `char`
- `checked`
- `class`
- `const`
- `continue`
- `decimal`
- `default`
- `delegate`
- `do`
- `double`
- `else`
- `enum`
- `event`
- `explicit`
- `extern`
- `false`
- `finally`
- `fixed`
- `float`
- `for`
- `foreach`
- `goto`
- `if`
- `implicit`
- `in`
- `int`
- `interface`
- `internal`
- `is`

- lock
- long
- namespace
- new
- null
- object
- operator
- out
- override
- params
- private
- protected
- public
- readonly
- ref
- return
- sbyte
- sealed
- short
- sizeof
- stackalloc
- static
- string
- struct
- switch
- this
- throw
- true
- try
- typeof
- uint
- ulong
- unchecked
- unsafe
- ushort
- using (direttiva)
- using (dichiarazione)
- virtual
- void
- volatile
- when
- while

Oltre a questi, C # utilizza anche alcune parole chiave per fornire un significato specifico nel codice. Sono chiamati parole chiave contestuali. Le parole chiave contestuali possono essere utilizzate come identificatori e non devono essere precedute da prefisso `@` quando vengono utilizzate come identificatori.

- add
- alias
- ascending
- async
- await
- descending
- dynamic

- from
- get
- global
- group
- into
- join
- let
- nameof
- orderby
- partial
- remove
- select
- set
- value
- var
- where
- yield

Examples

stackalloc

La parola chiave `stackalloc` crea una regione di memoria nello stack e restituisce un puntatore all'inizio di tale memoria. La memoria allocata nello stack viene automaticamente rimossa quando viene chiuso l'ambito in cui è stato creato.

```
//Allocate 1024 bytes. This returns a pointer to the first byte.
byte* ptr = stackalloc byte[1024];

//Assign some values...
ptr[0] = 109;
ptr[1] = 13;
ptr[2] = 232;
...
```

Utilizzato in un contesto non sicuro.

Come con tutti i puntatori in C # non ci sono limiti di controllo su letture e compiti. La lettura oltre i limiti della memoria allocata avrà risultati imprevedibili - potrebbe accedere a qualche posizione arbitraria all'interno della memoria o potrebbe causare un'eccezione di violazione di accesso.

```
//Allocate 1 byte
byte* ptr = stackalloc byte[1];

//Unpredictable results...
ptr[10] = 1;
ptr[-1] = 2;
```

La memoria allocata nello stack viene automaticamente rimossa quando viene chiuso l'ambito in cui è stato creato. Ciò significa che non si dovrebbe mai restituire la memoria creata con stackalloc o conservarla oltre la durata dell'ambito.

```
unsafe IntPtr Leak() {
    //Allocate some memory on the stack
    var ptr = stackalloc byte[1024];

    //Return a pointer to that memory (this exits the scope of "Leak")
    return new IntPtr(ptr);
}

unsafe void Bad() {
     //ptr is now an invalid pointer, using it in any way will have
     //unpredictable results. This is exactly the same as accessing beyond
     //the bounds of the pointer.
     var ptr = Leak();
}
```

`stackalloc` può essere usato solo quando si dichiarano *e si* inizializzano variabili. Quanto segue *non* è valido:

```
byte* ptr;
...
ptr = stackalloc byte[1024];
```

Osservazioni:

`stackalloc` deve essere usato solo per ottimizzare le prestazioni (sia per il calcolo che per l'interoperabilità). Ciò è dovuto al fatto che:

- Il garbage collector non è necessario in quanto la memoria viene allocata nello stack piuttosto che nell'heap - la memoria viene rilasciata non appena la variabile esce dallo scope
- È più veloce allocare memoria nello stack anziché nell'heap
- Aumentare la possibilità di colpi di cache sulla CPU a causa della località dei dati

volatile

L'aggiunta della parola chiave `volatile` a un campo indica al compilatore che il valore del campo può essere modificato da più thread separati. Lo scopo principale della parola chiave `volatile` è impedire le ottimizzazioni del compilatore che presuppongono solo l'accesso a thread singolo. L'utilizzo di `volatile` garantisce che il valore del campo sia il valore più recente disponibile e che il valore non sia soggetto alla memorizzazione nella cache dei valori non volatili.

È buona norma contrassegnare *tutte le variabili* che possono essere utilizzate da più thread come `volatile` per prevenire comportamenti imprevisti a causa di ottimizzazioni dietro le quinte. Considera il seguente blocco di codice:

```
public class Example
{
    public int x;

    public void DoStuff()
    {
        x = 5;
```

```
        // the compiler will optimize this to y = 15
        var y = x + 10;

        /* the value of x will always be the current value, but y will always be "15" */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

Nel suddetto codice, il compilatore legge le istruzioni `x = 5` `y = x + 10` e determina che il valore di `y` finirà sempre per 15. Quindi, ottimizzerà l'ultima istruzione come `y = 15` . Tuttavia, la variabile `x` è in realtà un campo `public` e il valore di `x` può essere modificato in fase di esecuzione attraverso un thread diverso che agisce su questo campo separatamente. Ora considera questo blocco di codice modificato. Si noti che il campo `x` è ora dichiarato come `volatile` .

```
public class Example
{
    public volatile int x;

    public void DoStuff()
    {
        x = 5;

        // the compiler no longer optimizes this statement
        var y = x + 10;

        /* the value of x and y will always be the correct values */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

Ora, il compilatore cerca gli usi di *lettura* del campo `x` e assicura che il valore corrente del campo sia sempre recuperato. Ciò garantisce che anche se più thread stanno leggendo e scrivendo in questo campo, il valore corrente di `x` viene sempre recuperato.

`volatile` può essere utilizzato solo su campi all'interno di `class` o `struct` . Quanto segue *non* è valido :

```
public void MyMethod()
{
    volatile int x;
}
```

`volatile` può essere applicato solo ai campi dei seguenti tipi:

- tipi di riferimento o parametri di tipo generico noti per essere tipi di riferimento
- tipi primitivi come `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `char` , `float` e `bool`
- tipi di enumerazione basati su `byte` , `sbyte` , `short` , `ushort` , `int` o `uint`
- `IntPtr` e `UIntPtr`

Osservazioni:

- Il modificatore `volatile` viene in genere utilizzato per un campo a cui si accede da più thread

senza utilizzare l'istruzione lock per serializzare l'accesso.
- La parola chiave `volatile` può essere applicata a campi di tipi di riferimento
- La parola chiave `volatile` non renderà operativo su primitive a 64 bit su una piattaforma atomica a 32 bit. Le operazioni interbloccate come `Interlocked.Read` e `Interlocked.Exchange` devono ancora essere utilizzate per un accesso multi-thread sicuro su queste piattaforme.

fisso

L'istruzione fissa corregge la memoria in un'unica posizione. Gli oggetti in memoria di solito si spostano, questo rende possibile la raccolta dei dati inutili. Ma quando usiamo i puntatori non sicuri agli indirizzi di memoria, quella memoria non deve essere spostata.

- Usiamo l'istruzione fixed per assicurare che il garbage collector non rilasci i dati di stringa.

Risolto Variabili

```
var myStr = "Hello world!";

fixed (char* ptr = myStr)
{
    // myStr is now fixed (won't be [re]moved by the Garbage Collector).
    // We can now do something with ptr.
}
```

Utilizzato in un contesto non sicuro.

Fixed Array Size

```
unsafe struct Example
{
    public fixed byte SomeField[8];
    public fixed char AnotherField[64];
}
```

`fixed` può essere usato solo sui campi di una `struct` (deve essere usato anche in un contesto non sicuro).

predefinito

Per classi, interfacce, delegate, array, `default(TheType) null` (come int?) E tipi di puntatore, il `default(TheType)` restituisce `null` :

```
class MyClass {}
Debug.Assert(default(MyClass) == null);
Debug.Assert(default(string) == null);
```

Per le strutture e le enumerazioni, il `default(TheType)` restituisce lo stesso del `new TheType()` :

```
struct Coordinates
{
```

```
    public int X { get; set; }
    public int Y { get; set; }
}

struct MyStruct
{
    public string Name { get; set; }
    public Coordinates Location { get; set; }
    public Coordinates? SecondLocation { get; set; }
    public TimeSpan Duration { get; set; }
}

var defaultStruct = default(MyStruct);
Debug.Assert(defaultStruct.Equals(new MyStruct()));
Debug.Assert(defaultStruct.Location.Equals(new Coordinates()));
Debug.Assert(defaultStruct.Location.X == 0);
Debug.Assert(defaultStruct.Location.Y == 0);
Debug.Assert(defaultStruct.SecondLocation == null);
Debug.Assert(defaultStruct.Name == null);
Debug.Assert(defaultStruct.Duration == TimeSpan.Zero);
```

`default(T)` può essere particolarmente utile quando `T` è un parametro generico per il quale non è presente alcun vincolo per decidere se `T` è un tipo di riferimento o un tipo di valore, ad esempio:

```
public T GetResourceOrDefault<T>(string resourceName)
{
   if (ResourceExists(resourceName))
   {
      return (T)GetResource(resourceName);
   }
   else
   {
      return default(T);
   }
}
```

sola lettura

La parola chiave `readonly` è un modificatore di campo. Quando una dichiarazione di campo include un modificatore di `readonly` , le assegnazioni a quel campo possono avvenire solo come parte della dichiarazione o in un costruttore della stessa classe.

La parola chiave `readonly` è diversa dalla parola chiave `const` . Un campo `const` può essere inizializzato solo alla dichiarazione del campo. Un campo di `readonly` può essere inizializzato o alla dichiarazione o in un costruttore. Pertanto, i campi di `readonly` possono avere valori diversi a seconda del costruttore utilizzato.

La parola chiave `readonly` viene spesso utilizzata quando si inseriscono le dipendenze.

```
class Person
{
    readonly string _name;
    readonly string _surname = "Surname";

    Person(string name)
```

```
    {
        _name = name;
    }
    void ChangeName()
    {
        _name = "another name"; // Compile error
        _surname = "another surname"; // Compile error
    }
}
```

> Nota: la dichiarazione di un campo in *sola lettura* non implica l' *immutabilità* . Se il campo è un *tipo di riferimento,* è possibile modificare il **contenuto** dell'oggetto. *In sola lettura* viene in genere utilizzato per impedire che l'oggetto venga **sovrascritto** e assegnato solo durante l' **istanziazione** di tale oggetto.
>
> Nota: all'interno del costruttore è possibile riassegnare un campo di sola lettura

```
public class Car
{
    public double Speed {get; set;}
}

//In code

private readonly Car car = new Car();

private void SomeMethod()
{
    car.Speed = 100;
}
```

come

La parola chiave `as` è un operatore simile a un *cast* . Se un cast non è possibile, l'utilizzo di `as` produce `null` che risultare in `InvalidCastException` .

`expression as type` è equivalente a `expression is type ? (type)expression : (type)null` con l'avvertenza che `as` è valido solo alle conversioni di riferimento, conversioni Null e conversioni boxe. Le conversioni definite dall'utente *non* sono supportate; al suo posto deve essere usato un cast regolare.

Per l'espansione di cui sopra, il compilatore genera codice tale che l' `expression` sarà valutata solo una volta e utilizzerà un controllo di tipo dinamico singolo (a differenza dei due nell'esempio sopra riportato).

`as` può essere utile quando si aspetta un argomento per facilitare diversi tipi. In particolare, concede all'utente più opzioni - piuttosto che controllare ogni possibilità con `is` prima del casting, o semplicemente lanciare e catturare le eccezioni. È consigliabile utilizzare "come" quando si esegue il casting / controllo di un oggetto che causerà solo una penalità di annullamento. L'utilizzo `is` di controllare, quindi il cast causerà due penalità di annullamento.

Se un argomento dovrebbe essere un'istanza di un tipo specifico, è preferibile un cast regolare in

quanto il suo scopo è più chiaro per il lettore.

Poiché una chiamata a `as` può produrre `null` , controllare sempre il risultato di evitare un `NullReferenceException` .

Esempio di utilizzo

```
object something = "Hello";
Console.WriteLine(something as string);        //Hello
Console.Writeline(something as Nullable<int>); //null
Console.WriteLine(something as int?);          //null

//This does NOT compile:
//destination type must be a reference type (or a nullable value type)
Console.WriteLine(something as int);
```

Live Demo su .NET Fiddle

Esempio equivalente senza utilizzo `as` :

```
Console.WriteLine(something is string ? (string)something : (string)null);
```

Ciò è utile quando si esegue l'override della funzione `Equals` nelle classi personalizzate.

```
class MyCustomClass
{

    public override bool Equals(object obj)
    {
        MyCustomClass customObject = obj as MyCustomClass;

        // if it is null it may be really null
        // or it may be of a different type
        if (Object.ReferenceEquals(null, customObject))
        {
            // If it is null then it is not equal to this instance.
            return false;
        }

        // Other equality controls specific to class
    }

}
```

è

Verifica se un oggetto è compatibile con un determinato tipo, cioè se un oggetto è un'istanza del tipo `BaseInterface` o un tipo che deriva da `BaseInterface` :

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
```

```
Console.WriteLine(d is DerivedClass);  // True
Console.WriteLine(d is BaseClass);     // True
Console.WriteLine(d is BaseInterface); // True
Console.WriteLine(d is object);        // True
Console.WriteLine(d is string);        // False

var b = new BaseClass();
Console.WriteLine(b is DerivedClass);  // False
Console.WriteLine(b is BaseClass);     // True
Console.WriteLine(b is BaseInterface); // True
Console.WriteLine(b is object);        // True
Console.WriteLine(b is string);        // False
```

Se l'intento del cast è quello di utilizzare l'oggetto, è buona norma utilizzare il `as` parola chiave'

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass);  // True - valid use of 'is'
Console.WriteLine(d is BaseClass);     // True - valid use of 'is'

if(d is BaseClass){
    var castedD = (BaseClass)d;
    castedD.Method(); // valid, but not best practice
}

var asD = d as BaseClass;

if(asD!=null){
    asD.Method(); //prefered method since you incur only one unboxing penalty
}
```

Tuttavia, dalla caratteristica di `pattern matching` C # 7 si estende l'operatore is per controllare un tipo e dichiarare una nuova variabile allo stesso tempo. La stessa parte di codice con C # 7:

7.0

```
if(d is BaseClass asD ){
    asD.Method();
}
```

tipo di

Restituisce il `Type` di un oggetto, senza bisogno di istanziarlo.

```
Type type = typeof(string);
Console.WriteLine(type.FullName); //System.String
Console.WriteLine("Hello".GetType() == type); //True
Console.WriteLine("Hello".GetType() == typeof(string)); //True
```

const

`const` è usato per rappresentare valori che **non cambieranno mai per** tutta la durata del programma. Il suo valore è costante dalla **fase** di **compilazione** , a differenza della parola chiave `readonly` , il cui valore è costante dal tempo di esecuzione.

Ad esempio, poiché la velocità della luce non cambierà mai, possiamo memorizzarla in una costante.

```
const double c = 299792458;  // Speed of light

double CalculateEnergy(double mass)
{
    return mass * c * c;
}
```

Questo è essenzialmente uguale alla `return mass * 299792458 * 299792458` , in quanto il compilatore sostituirà direttamente `c` con il suo valore costante.

Di conseguenza, `c` non può essere modificato una volta dichiarato. Quanto segue produrrà un errore in fase di compilazione:

```
const double c = 299792458;  // Speed of light

c = 500;  //compile-time error
```

Una costante può essere preceduta dagli stessi modificatori di accesso dei metodi:

```
private const double c = 299792458;
public const double c = 299792458;
internal const double c = 299792458;
```

`const` membri `const` sono `static` per natura. Tuttavia, l'uso `static` esplicito non è permesso.

È inoltre possibile definire le costanti del metodo locale:

```
double CalculateEnergy(double mass)
{
    const c = 299792458;
    return mass * c * c;
}
```

Questi non possono essere preceduti da una parola chiave `private` o `public` , poiché sono implicitamente locali rispetto al metodo in cui sono definiti.

Non tutti i tipi possono essere utilizzati in una dichiarazione `const` . I tipi di valori consentiti sono i tipi predefiniti `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` , `bool` e tutti i tipi `enum` . Provare a dichiarare membri `const` con altri tipi di valore (come `TimeSpan` o `Guid`) fallirà in fase di compilazione.

Per la speciale `string` tipo di riferimento predefinita, le costanti possono essere dichiarate con qualsiasi valore. Per tutti gli altri tipi di riferimento, le costanti possono essere dichiarate ma

devono sempre avere il valore `null` .

Poiché i valori `const` sono noti in fase di compilazione, sono consentiti come etichette `case` in un'istruzione `switch` , come argomenti standard per parametri facoltativi, come argomenti per l'attribuzione di specifiche e così via.

Se i valori `const` vengono utilizzati tra diversi assembly, è necessario prestare attenzione con il controllo delle versioni. Ad esempio, se l'assembly A definisce un `public const int MaxRetries = 3;` e l'assembly B usa quella costante, quindi se il valore di `MaxRetries` viene in seguito modificato in `5` nell'assembly A (che viene quindi ricompilato), tale modifica non sarà effettiva nell'assembly B a *meno che l'* assembly B non venga ricompilato (con un riferimento alla nuova versione di A).

Per questo motivo, se un valore può cambiare nelle revisioni future del programma e se il valore deve essere visibile pubblicamente, non dichiarare tale valore `const` se non si è certi che tutti gli assembly dipendenti verranno ricompilati ogni volta che qualcosa viene modificato. L'alternativa sta usando `static readonly` invece di `const` , che viene risolto in fase di runtime.

namespace

La parola chiave `namespace` è un costrutto organizzativo che ci aiuta a capire come è organizzato un codebase. I namespace in C # sono spazi virtuali piuttosto che essere in una cartella fisica.

```
namespace StackOverflow
{
    namespace Documentation
    {
        namespace CSharp.Keywords
        {
            public class Program
            {
                public static void Main()
                {
                    Console.WriteLine(typeof(Program).Namespace);
                    //StackOverflow.Documentation.CSharp.Keywords
                }
            }
        }
    }
}
```

I namespace in C # possono anche essere scritti in sintassi concatenata. Quanto segue è equivalente a sopra:

```
namespace StackOverflow.Documentation.CSharp.Keywords
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine(typeof(Program).Namespace);
            //StackOverflow.Documentation.CSharp.Keywords
```

```
        }
    }
}
```

prova, prendi, finalmente, lancia

`try` , `catch` , `finally` e `throw` ti permettono di gestire le eccezioni nel tuo codice.

```
var processor = new InputProcessor();

// The code within the try block will be executed. If an exception occurs during execution of
// this code, execution will pass to the catch block corresponding to the exception type.
try
{
    processor.Process(input);
}
// If a FormatException is thrown during the try block, then this catch block
// will be executed.
catch (FormatException ex)
{
    // Throw is a keyword that will manually throw an exception, triggering any catch block
that is
    // waiting for that exception type.
    throw new InvalidOperationException("Invalid input", ex);
}
// catch can be used to catch all or any specific exceptions. This catch block,
// with no type specified, catches any exception that hasn't already been caught
// in a prior catch block.
catch
{
    LogUnexpectedException();
    throw; // Re-throws the original exception.
}
// The finally block is executed after all try-catch blocks have been; either after the try
has
// succeeded in running all commands or after all exceptions have been caught.
finally
{
    processor.Dispose();
}
```

Nota: la parola chiave `return` può essere utilizzata nel blocco `try` e il blocco `finally` verrà comunque eseguito (appena prima di tornare). Per esempio:

```
try
{
    connection.Open();
    return connection.Get(query);
}
finally
{
    connection.Close();
}
```

L'istruzione `connection.Close()` verrà eseguita prima del risultato della `connection.Get(query)` viene restituito.

Continua

Passa immediatamente il controllo alla successiva iterazione del costrutto del loop che lo racchiude (for, foreach, do, while):

```
for (var i = 0; i < 10; i++)
{
    if (i < 5)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

Produzione:

5
6
7
8
9

Live Demo su .NET Fiddle

```
var stuff = new [] {"a", "b", null, "c", "d"};

foreach (var s in stuff)
{
    if (s == null)
    {
        continue;
    }
    Console.WriteLine(s);
}
```

Produzione:

un
B
c
d

Live Demo su .NET Fiddle

ref, fuori

Le parole chiave `ref` e `out` sì che un argomento venga passato per riferimento, non per valore. Per i tipi di valore, ciò significa che il valore della variabile può essere modificato dal destinatario.

```
int x = 5;
ChangeX(ref x);
// The value of x could be different now
```

Per i tipi di riferimento, l'istanza nella variabile non può essere modificata solo (come nel caso senza `ref`), ma può anche essere sostituita del tutto:

```
Address a = new Address();
ChangeFieldInAddress(a);
// a will be the same instance as before, even if it is modified
CreateANewInstance(ref a);
// a could be an entirely new instance now
```

La principale differenza tra la parola chiave `out` e `ref` è che `ref` richiede che la variabile venga inizializzata dal chiamante, mentre `out` passa tale responsabilità al destinatario.

Per utilizzare un parametro `out` , sia la definizione del metodo sia il metodo di chiamata devono utilizzare esplicitamente la parola chiave `out` .

```
int number = 1;
Console.WriteLine("Before AddByRef: " + number); // number = 1
AddOneByRef(ref number);
Console.WriteLine("After AddByRef: " + number);  // number = 2
SetByOut(out number);
Console.WriteLine("After SetByOut: " + number);  // number = 34

void AddOneByRef(ref int value)
{
    value++;
}

void SetByOut(out int value)
{
    value = 34;
}
```

Live Demo su .NET Fiddle

Quanto segue non *si* compila, perché `out` parametri `out` devono avere un valore assegnato prima che il metodo ritorni (si compilerebbe invece usando `ref`):

```
void PrintByOut(out int value)
{
    Console.WriteLine("Hello!");
}
```

usando la parola chiave come modificatore generico

`out` parola chiave `out` può essere utilizzata anche nei parametri di tipo generico quando si definiscono interfacce e delegati generici. In questo caso, la parola chiave `out` specifica che il parametro type è covariante.

> La covarianza consente di utilizzare un tipo più derivato rispetto a quello specificato dal parametro generico. Ciò consente la conversione implicita di classi che implementano interfacce varianti e la conversione implicita di tipi di delegati. Covarianza e controvarianza sono supportate per i tipi di riferimento, ma non sono supportate per i tipi di valore. - MSDN

```
//if we have an interface like this
interface ICovariant<out R> { }

//and two variables like
ICovariant<Object> iobj = new Sample<Object>();
ICovariant<String> istr = new Sample<String>();

// then the following statement is valid
// without the out keyword this would have thrown error
iobj = istr; // implicit conversion occurs here
```

selezionato, deselezionato

Le parole chiave `checked` e `unchecked` definiscono come le operazioni gestiscono l'overflow matematico. "Overflow" nel contesto delle parole chiave `checked` e `unchecked` si verifica quando un'operazione di aritmetica intera dà come risultato un valore maggiore di grandezza rispetto a quello che il tipo di dati di destinazione può rappresentare.

Quando si verifica un overflow all'interno di un blocco `checked` (o quando il compilatore è impostato per utilizzare globalmente l'aritmetica controllata), viene lanciata un'eccezione per avvisare di un comportamento indesiderato. Nel frattempo, in un blocco `unchecked` , l'overflow è silenzioso: non vengono generate eccezioni e il valore si limita semplicemente al limite opposto. Questo può portare a bug sottili e difficili da trovare.

Dal momento che la maggior parte delle operazioni aritmetiche vengono eseguite su valori che non sono grandi o abbastanza piccoli da eccedere, la maggior parte delle volte non è necessario definire esplicitamente un blocco come `checked` . È necessario prestare attenzione quando si eseguono operazioni aritmetiche su input non limitati che possono causare un overflow, ad esempio quando si esegue l'aritmetica in funzioni ricorsive o mentre si immette l'input dell'utente.

Né `checked` né `unchecked` influiscono sulle operazioni aritmetiche in virgola mobile.

Quando un blocco o un'espressione viene dichiarato come `unchecked` , qualsiasi operazione aritmetica al suo interno può eccedere senza causare errori. Un esempio in cui questo comportamento è *desiderato* potrebbe essere il calcolo di un checksum, in cui il valore è consentito per "avvolgere" durante il calcolo:

```
byte Checksum(byte[] data) {
    byte result = 0;
    for (int i = 0; i < data.Length; i++) {
        result = unchecked(result + data[i]); // unchecked expression
    }
    return result;
}
```

Uno degli usi più comuni per `unchecked` è l'implementazione di una sovrascrittura personalizzata per `object.GetHashCode()` , un tipo di checksum. Puoi vedere l'uso della parola chiave nelle risposte a questa domanda: qual è il miglior algoritmo per un System.Object.GetHashCode sottoposto a override? .

Quando un blocco o un'espressione viene dichiarata come `checked` , qualsiasi operazione

aritmetica che provoca un overflow genera una `OverflowException` generata.

```
int SafeSum(int x, int y) {
    checked { // checked block
        return x + y;
    }
}
```

Sia selezionato che deselezionato possono essere in forma di blocco ed espressione.

I blocchi selezionati e deselezionati non influiscono sui metodi chiamati, solo gli operatori chiamati direttamente nel metodo corrente. Ad esempio, `Enum.ToObject()` , `Convert.ToInt32()` e gli operatori definiti dall'utente non sono interessati dai contesti personalizzati selezionati / non selezionati.

***Nota** : il comportamento predefinito di overflow predefinito (selezionato o deselezionato) può essere modificato nelle proprietà del **progetto** o tramite l' **opzione della** riga di comando **/ controllata [+ | -]** . È normale eseguire il controllo delle operazioni di debug e deselezionare le build di rilascio. Le parole chiave `checked` e `unchecked` verranno quindi utilizzate solo laddove non si applica un approccio predefinito e per garantire la correttezza è necessario un comportamento esplicito.*

vai a

`goto` può essere utilizzato per passare a una riga specifica all'interno del codice, specificata da un'etichetta.

`goto` come:

Etichetta:

```
void InfiniteHello()
{
    sayHello:
    Console.WriteLine("Hello!");
    goto sayHello;
}
```

Live Demo su .NET Fiddle

Caso clinico:

```
enum Permissions { Read, Write };

switch (GetRequestedPermission())
{
    case Permissions.Read:
        GrantReadAccess();
        break;
```

```
    case Permissions.Write:
        GrantWriteAccess();
        goto case Permissions.Read; //People with write access also get read
}
```

Live Demo su .NET Fiddle

Ciò è particolarmente utile nell'esecuzione di più comportamenti in un'istruzione switch, in quanto C # non supporta i blocchi di casi fall-through .

Eccezione Riprova

```
var exCount = 0;
retry:
try
{
    //Do work
}
catch (IOException)
{
    exCount++;
    if (exCount < 3)
    {
        Thread.Sleep(100);
        goto retry;
    }
    throw;
}
```

Live Demo su .NET Fiddle

Simile a molte lingue, l'uso della parola chiave goto è sconsigliato tranne i casi di seguito.

Validi usi di `goto` che si applicano a C #:

- Caso fall-through nell'istruzione switch.
- Pausa multi-livello LINQ può essere usato spesso, ma di solito ha prestazioni peggiori.
- Deallocazione delle risorse quando si lavora con oggetti di basso livello non aperti. In C #, gli oggetti di basso livello dovrebbero essere normalmente racchiusi in classi separate.
- Macchine a stati finiti, ad esempio parser; utilizzato internamente dal compilatore generato async / await state machine.

enum

La parola chiave `enum` dice al compilatore che questa classe eredita dalla classe astratta `Enum` , senza che il programmatore debba ereditarlo esplicitamente. `Enum` è un discendente di `ValueType` , che è inteso per l'uso con un insieme distinto di costanti con nome.

```
public enum DaysOfWeek
{
    Monday,
    Tuesday,
}
```

Puoi facoltativamente specificare un valore specifico per ognuno (o alcuni di essi):

```
public enum NotableYear
{
   EndOfWwI = 1918;
   EnfOfWwII = 1945,
}
```

In questo esempio ho omesso un valore per 0, di solito è una cattiva pratica. Un `enum` avrà sempre un valore predefinito prodotto dalla conversione esplicita `(YourEnumType) 0` , dove `YourEnumType` è il tipo `enume` dichiarato. Senza un valore di 0 definito, un `enum` non avrà un valore definito all'avvio.

Il tipo di `enum` predefinito di base è `int` , è possibile modificare il tipo sottostante a qualsiasi tipo integrale incluso `byte` , `sbyte` , `short` , `ushort` , `int` , `uint` , `long` e `ulong` . Di seguito è riportato un enum con `byte` tipo sottostante:

```
enum Days : byte
{
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

Si noti inoltre che è possibile convertire in / dal tipo sottostante semplicemente con un cast:

```
int value = (int)NotableYear.EndOfWwI;
```

Per questi motivi è meglio controllare sempre se un `enum` è valido quando si espongono le funzioni della libreria:

```
void PrintNotes(NotableYear year)
{
    if (!Enum.IsDefined(typeof(NotableYear), year))
        throw InvalidEnumArgumentException("year", (int)year, typeof(NotableYear));

    // ...
}
```

base

La parola chiave di **`base`** viene utilizzata per accedere ai membri da una classe base. Viene comunemente utilizzato per chiamare le implementazioni di base dei metodi virtuali o per

specificare quale costruttore di base deve essere chiamato.

Scegliere un costruttore

```
public class Child : SomeBaseClass {
    public Child() : base("some string for the base class")
    {
    }
}

public class SomeBaseClass {
    public SomeBaseClass()
    {
        // new Child() will not call this constructor, as it does not have a parameter
    }
    public SomeBaseClass(string message)
    {
        // new Child() will use this base constructor because of the specified parameter in
Child's constructor
        Console.WriteLine(message);
    }
}
```

Chiamare l'implementazione di base del metodo virtuale

```
public override void SomeVirtualMethod() {
    // Do something, then call base implementation
    base.SomeVirtualMethod();
}
```

È possibile utilizzare la parola chiave di base per chiamare un'implementazione di base da qualsiasi metodo. Ciò lega la chiamata del metodo direttamente all'implementazione di base, il che significa che anche se le nuove classi secondarie sostituiscono un metodo virtuale, l'implementazione di base verrà comunque chiamata, quindi è necessario utilizzarla con cautela.

```
public class Parent
{
    public virtual int VirtualMethod()
    {
        return 1;
    }
}

public class Child : Parent
{
    public override int VirtualMethod() {
        return 11;
    }

    public int NormalMethod()
    {
        return base.VirtualMethod();
    }

    public void CallMethods()
    {
        Assert.AreEqual(11, VirtualMethod());
```

```
        Assert.AreEqual(1, NormalMethod());
        Assert.AreEqual(1, base.VirtualMethod());
    }
}

public class GrandChild : Child
{
    public override int VirtualMethod()
    {
        return 21;
    }

    public void CallAgain()
    {
        Assert.AreEqual(21, VirtualMethod());
        Assert.AreEqual(11, base.VirtualMethod());

        // Notice that the call to NormalMethod below still returns the value
        // from the extreme base class even though the method has been overridden
        // in the child class.
        Assert.AreEqual(1, NormalMethod());
    }
}
```

per ciascuno

`foreach` viene utilizzato per scorrere gli elementi di una matrice o gli elementi all'interno di una raccolta che implementa `IEnumerable` †.

```
var lines = new string[] {
    "Hello world!",
    "How are you doing today?",
    "Goodbye"
};

foreach (string line in lines)
{
    Console.WriteLine(line);
}
```

Questo uscirà

> "Ciao mondo!"
> "Come stai oggi?"
> "Addio"

Live Demo su .NET Fiddle

È possibile uscire dal ciclo `foreach` in qualsiasi momento utilizzando la parola chiave break o passare alla successiva iterazione utilizzando la parola chiave continue .

```
var numbers = new int[] {1, 2, 3, 4, 5, 6};

foreach (var number in numbers)
```

```
{
    // Skip if 2
    if (number == 2)
        continue;

    // Stop iteration if 5
    if (number == 5)
        break;

    Console.Write(number + ", ");
}

// Prints: 1, 3, 4,
```

Live Demo su .NET Fiddle

Si noti che l'ordine di iterazione è garantito *solo* per determinate raccolte come matrici ed `List` , ma **non è** garantito per molte altre raccolte.

---

† Mentre `IEnumerable` viene in genere utilizzato per indicare le raccolte enumerabili, `foreach` richiede solo che la raccolta esponga pubblicamente il metodo `object GetEnumerator()` , che deve restituire un oggetto che espone il metodo `bool MoveNext()` e l' `object Current { get; }` proprietà.

params

`params` consente a un parametro del metodo di ricevere un numero variabile di argomenti, vale a dire zero, uno o più argomenti sono consentiti per quel parametro.

```
static int AddAll(params int[] numbers)
{
    int total = 0;
    foreach (int number in numbers)
    {
        total += number;
    }

    return total;
}
```

Questo metodo può ora essere chiamato con una lista tipica di argomenti `int` , o una matrice di interi.

```
AddAll(5, 10, 15, 20);                // 50
AddAll(new int[] { 5, 10, 15, 20 });  // 50
```

`params` deve apparire al massimo una volta e se usato, deve essere l' **ultimo** nella lista degli argomenti, anche se il tipo successivo è diverso da quello dell'array.

Fare attenzione quando si sovraccaricano le funzioni quando si utilizza la parola chiave `params` . C # preferisce abbinare sovraccarichi più specifici prima di ricorrere al tentativo di usare sovraccarichi con `params` . Ad esempio se hai due metodi:

```
static double Add(params double[] numbers)
{
    Console.WriteLine("Add with array of doubles");
    double total = 0.0;
    foreach (double number in numbers)
    {
        total += number;
    }

    return total;
}

static int Add(int a, int b)
{
    Console.WriteLine("Add with 2 ints");
    return a + b;
}
```

Quindi lo specifico sovraccarico di 2 argomenti avrà la precedenza prima di provare il sovraccarico dei `params` .

```
Add(2, 3);      //prints "Add with 2 ints"
Add(2, 3.0);    //prints "Add with array of doubles" (doubles are not ints)
Add(2, 3, 4);   //prints "Add with array of doubles" (no 3 argument overload)
```

rompere

In un ciclo (for, foreach, do, while) l'istruzione `break` interrompe l'esecuzione del ciclo più interno e ritorna al codice dopo di esso. Inoltre può essere utilizzato con `yield` in cui specifica che un iteratore è giunto al termine.

```
for (var i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break;
    }
    Console.WriteLine("This will appear only 5 times, as the break will stop the loop.");
}
```

Live Demo su .NET Fiddle

```
foreach (var stuff in stuffCollection)
{
    if (stuff.SomeStringProp == null)
        break;
    // If stuff.SomeStringProp for any "stuff" is null, the loop is aborted.
    Console.WriteLine(stuff.SomeStringProp);
}
```

L'istruzione break viene anche utilizzata nei costrutti del caso di commutazione per uscire da un caso o da un segmento predefinito.

```
switch(a)
```

```
{
    case 5:
        Console.WriteLine("a was 5!");
        break;

    default:
        Console.WriteLine("a was something else!");
        break;
}
```

Nelle istruzioni switch, la parola chiave 'break' è richiesta alla fine di ogni dichiarazione di caso. Ciò è contrario ad alcune lingue che consentono di "passare attraverso" alla successiva dichiarazione del caso nella serie. I rimedi per questo includevano le istruzioni "goto" o l'impilamento sequenziale delle dichiarazioni "case".

Il codice seguente darà i numeri `0, 1, 2, ..., 9` e l'ultima riga non verrà eseguita. `yield break` indica la fine della funzione (non solo un loop).

```
public static IEnumerable<int> GetNumbers()
{
    int i = 0;
    while (true) {
        if (i < 10) {
            yield return i++;
        } else {
            yield break;
        }
    }
    Console.WriteLine("This line will not be executed");
}
```

Live Demo su .NET Fiddle

Nota che a differenza di altri linguaggi, non c'è modo di etichettare una particolare interruzione in C #. Ciò significa che nel caso di cicli annidati, verrà interrotto solo il ciclo più interno:

```
foreach (var outerItem in outerList)
{
    foreach (var innerItem in innerList)
    {
        if (innerItem.ShoudBreakForWhateverReason)
            // This will only break out of the inner loop, the outer will continue:
            break;
    }
}
```

Se vuoi uscire dal ciclo *esterno* qui, puoi utilizzare una delle diverse strategie, ad esempio:

- Una dichiarazione **goto** per saltare fuori dall'intera struttura del ciclo.
- Una specifica variabile di flag (`shouldBreak` nell'esempio seguente) che può essere verificata alla fine di ogni iterazione del ciclo esterno.
- Rifattorizzare il codice per utilizzare un'istruzione `return` nel corpo del ciclo più interno o evitare del tutto l'intera struttura del ciclo annidato.

```
bool shouldBreak = false;
while(comeCondition)
{
    while(otherCondition)
    {
        if (conditionToBreak)
        {
            // Either tranfer control flow to the label below...
            goto endAllLooping;

            // OR use a flag, which can be checked in the outer loop:
            shouldBreak = true;
        }
    }

    if(shouldBreakNow)
    {
        break; // Break out of outer loop if flag was set to true
    }
}

endAllLooping: // label from where control flow will continue
```

astratto

Una classe contrassegnata con la parola chiave `abstract` non può essere istanziata.

Una classe *deve* essere contrassegnata come astratta se contiene membri astratti o se eredita membri astratti che non implementa. Una classe *può* essere contrassegnata come astratta anche se non sono coinvolti membri astratti.

Le classi astratte vengono solitamente utilizzate come classi base quando alcune parti dell'implementazione devono essere specificate da un altro componente.

```
abstract class Animal
{
    string Name { get; set; }
    public abstract void MakeSound();
}

public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meov meov");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bark bark");
    }
}

Animal cat = new Cat();       // Allowed due to Cat deriving from Animal
```

```
cat.MakeSound();               // will print out "Meov meov"

Animal dog = new Dog();        // Allowed due to Dog deriving from Animal
dog.MakeSound();               // will print out "Bark bark"

Animal animal = new Animal(); // Not allowed due to being an abstract class
```

Un metodo, una proprietà o un evento contrassegnati con la parola chiave `abstract` indica che l'implementazione per tale membro dovrebbe essere fornita in una sottoclasse. Come accennato in precedenza, i membri astratti possono apparire solo in classi astratte.

```
abstract class Animal
{
   public abstract string Name { get; set; }
}

public class Cat : Animal
{
    public override string Name { get; set; }
}

public class Dog : Animal
{
    public override string Name { get; set; }
}
```

float, double, decimal

galleggiante

`float` è un alias per il tipo di dati .NET `System.Single` . Consente di memorizzare i numeri in virgola mobile a precisione singola IEEE 754. Questo tipo di dati è presente in `mscorlib.dll` cui fa riferimento implicitamente ogni progetto C # quando vengono creati.

Intervallo approssimativo: da -3,4 × 10 ^{38} a 3,4 × 10 ^{38}

Precisione decimale: 6-9 cifre significative

Notazione :

```
float f = 0.1259;
var f1 = 0.7895f; // f is literal suffix to represent float values
```

> Va notato che il tipo `float` spesso causa errori di arrotondamento significativi. Nelle applicazioni in cui la precisione è importante, devono essere considerati altri tipi di dati.

Doppio

`double`

è un alias del tipo di dati .NET `System.Double` . Rappresenta un numero a virgola mobile a 64 bit a doppia precisione. Questo tipo di dati è presente in `mscorlib.dll` cui viene fatto implicitamente riferimento in qualsiasi progetto C #.

Intervallo: ± 5,0 × 10 $^{-324}$ a ± 1,7 × 10 ^{308}

Precisione decimale: 15-16 cifre significative

Notazione :

```
double distance = 200.34; // a double value
double salary = 245; // an integer implicitly type-casted to double value
var marks = 123.764D; // D is literal suffix to represent double values
```

---

decimale

`decimal` è un alias del tipo di dati .NET `System.Decimal` . Rappresenta una parola chiave indica un tipo di dati a 128 bit. Rispetto ai tipi a virgola mobile, il tipo decimale ha più precisione e un intervallo più piccolo, il che lo rende appropriato per i calcoli finanziari e monetari. Questo tipo di dati è presente in `mscorlib.dll` cui viene fatto implicitamente riferimento in qualsiasi progetto C #.

Intervallo: da -7,9 × 10 ^{28} a 7,9 × 10 ^{28}

Precisione decimale: 28-29 cifre significative

Notazione :

```
decimal payable = 152.25m; // a decimal value
var marks = 754.24m; // m is literal suffix to represent decimal values
```

uint

Un **intero senza segno** , o **uint** , è un tipo di dati numerico che può contenere solo interi positivi. Come suggerisce il nome, rappresenta un numero intero a 32 bit senza segno. La stessa parola chiave **uint** è un alias per il tipo di sistema di tipo comune `System.UInt32` . Questo tipo di dati è presente in `mscorlib.dll` , a cui fa riferimento implicitamente ogni progetto C # quando vengono creati. Occupa quattro byte di spazio di memoria.

Gli interi senza segno possono contenere qualsiasi valore compreso tra 0 e 4.294.967.295.

Esempi su come e ora non dichiarare numeri interi senza segno

```
uint i = 425697; // Valid expression, explicitly stated to compiler
var i1 = 789247U; // Valid expression, suffix allows compiler to determine datatype
uint x = 3.0; // Error, there is no implicit conversion
```

Nota: secondo Microsoft , si consiglia di utilizzare il tipo di dati **int** ove possibile in quanto il tipo di

dati **uint** non è conforme a CLS.

Questo

Il `this` parola chiave si riferisce all'istanza corrente di classe (oggetto). In questo modo si possono distinguere due variabili con lo stesso nome, una a livello di classe (un campo) e una essendo un parametro (o una variabile locale) di un metodo.

```
public MyClass {
    int a;

    void set_a(int a)
    {
        //this.a refers to the variable defined outside of the method,
        //while a refers to the passed parameter.
        this.a = a;
    }
}
```

Altri usi della parola chiave sono concatenamento di sovraccarichi di costruttori non statici :

```
public MyClass(int arg) : this(arg, null)
{
}
```

e scrivere gli indicizzatori :

```
public string this[int idx1, string idx2]
{
    get { /* ... */ }
    set { /* ... */ }
}
```

e dichiarando i metodi di estensione :

```
public static int Count<TItem>(this IEnumerable<TItem> source)
{
    // ...
}
```

Se non v'è alcun conflitto con una variabile locale o un parametro, si tratta di una questione di stile se utilizzare `this` o no, così `this.MemberOfType` e `MemberOfType` sarebbero equivalenti in quel caso. Vedi anche la parola chiave di `base` .

Si noti che se un metodo di estensione deve essere chiamato in questa istanza, `this` è richiesto. Ad esempio se ci si trova all'interno di un metodo non statico di una classe che implementa `IEnumerable<>` e si desidera chiamare il `Count` dell'estensione di prima, è necessario utilizzare:

```
this.Count()  // works like StaticClassForExtensionMethod.Count(this)
```

e `this` non può essere omesso lì.

per

Sintassi: `for (initializer; condition; iterator)`

- Il ciclo `for` è comunemente usato quando è noto il numero di iterazioni.
- Le istruzioni nella sezione di `initializer` eseguite una sola volta, prima di entrare nel ciclo.
- La sezione `condition` contiene un'espressione booleana valutata alla fine di ogni iterazione del ciclo per determinare se il ciclo deve uscire o deve essere eseguito nuovamente.
- La sezione `iterator` definisce cosa succede dopo ogni iterazione del corpo del loop.

Questo esempio mostra come `for` può essere usato per scorrere i caratteri di una stringa:

```
string str = "Hello";
for (int i = 0; i < str.Length; i++)
{
    Console.WriteLine(str[i]);
}
```

Produzione:

H
e
l
l
o

Live Demo su .NET Fiddle

Tutte le espressioni che definiscono una dichiarazione `for` sono facoltative; ad esempio, la seguente istruzione è usata per creare un ciclo infinito:

```
for( ; ; )
{
    // Your code here
}
```

La sezione di `initializer` può contenere più variabili, purché siano dello stesso tipo. La sezione delle `condition` può essere costituita da qualsiasi espressione che può essere valutata da un `bool` . E la sezione `iterator` può eseguire più azioni separate da una virgola:

```
string hello = "hello";
for (int i = 0, j = 1, k = 9; i < 3 && k > 0; i++, hello += i) {
    Console.WriteLine(hello);
}
```

Produzione:

Ciao
hello1
hello12

Live Demo su .NET Fiddle

mentre

L'operatore `while` esegue iterazioni su un blocco di codice fino a quando la query condizionale è uguale a false o il codice viene interrotto con un'istruzione `goto` , `return` , `break` o `throw` .

Sintassi per parola chiave `while` :

> while (*condizione*) { *blocco di codice;* }

Esempio:

```
int i = 0;
while (i++ < 5)
{
    Console.WriteLine("While is on loop number {0}.", i);
}
```

Produzione:

> "Mentre è sul loop numero 1."
> "Mentre è sul loop numero 2."
> "Mentre è sul loop numero 3."
> "Mentre è sul loop numero 4."
> "Mentre è sul loop numero 5."

Live Demo su .NET Fiddle

Un ciclo while è **Entry Controlled** , poiché la condizione viene verificata **prima** dell'esecuzione del blocco di codice allegato. Ciò significa che il ciclo while non eseguirà le sue istruzioni se la condizione è falsa.

```
bool a = false;

while (a == true)
{
    Console.WriteLine("This will never be printed.");
}
```

Dare una condizione `while` senza provvedere a renderlo falso ad un certo punto si tradurrà in un ciclo infinito o infinito. Per quanto possibile, questo dovrebbe essere evitato, tuttavia, ci possono essere alcune circostanze eccezionali quando ne hai bisogno.

È possibile creare un loop come segue:

```
while (true)
{
//...
}
```

Si noti che il compilatore C # trasformerà loop come

```
while (true)
{
// ...
}
```

o

```
for(;;)
{
// ...
}
```

in

```
{
:label
// ...
goto label;
}
```

Si noti che un ciclo while può avere qualsiasi condizione, non importa quanto complessa, purché valuti (o restituisca) un valore booleano (bool). Può anche contenere una funzione che restituisce un valore booleano (in quanto tale una funzione valuta lo stesso tipo di un'espressione come `a == x '). Per esempio,

```
while (AgriculturalService.MoreCornToPick(myFarm.GetAddress()))
{
    myFarm.PickCorn();
}
```

ritorno

> MSDN: l'istruzione return termina l'esecuzione del metodo in cui appare e restituisce il controllo al metodo di chiamata. Può anche restituire un valore opzionale. Se il metodo è di tipo vuoto, l'istruzione return può essere omessa.

```
public int Sum(int valueA, int valueB)
{
    return valueA + valueB;
}


public void Terminate(bool terminateEarly)
{
    if (terminateEarly) return; // method returns to caller if true was passed in
    else Console.WriteLine("Not early"); // prints only if terminateEarly was false
}
```

nel

L' `in` parola chiave ha tre usi:

a) Come parte della sintassi in un'istruzione `foreach` o come parte della sintassi in una query LINQ

```
foreach (var member in sequence)
{
    // ...
}
```

b) Nel contesto di interfacce generiche e tipi di delegati generici indica la *controvarianza* per il parametro di tipo in questione:

```
public interface IComparer<in T>
{
    // ...
}
```

c) Nel contesto della query LINQ fa riferimento alla raccolta che viene interrogata

```
var query = from x in source select new { x.Name, x.ID, };
```

utilizzando

Esistono due tipi di `using` dell'utilizzo di parole chiave, `using statement` e `using directive` :

1. **usando la dichiarazione** :

   La parola chiave `using` assicura che gli oggetti che implementano l'interfaccia `IDisposable` siano disposti correttamente dopo l'uso. C'è un argomento separato per l' istruzione using

2. **usando la direttiva**

   La direttiva `using` ha tre usi, vedere la pagina msdn per la direttiva using . C'è un argomento separato per la direttiva using .

sigillato

Se applicato a una classe, il modificatore `sealed` impedisce ad altre classi di ereditarlo.

```
class A { }
sealed class B : A { }
class C : B { } //error : Cannot derive from the sealed class
```

Quando applicato a un metodo `virtual` (o proprietà virtuale), il modificatore `sealed` impedisce che questo metodo (proprietà) venga *sovrascritto* nelle classi derivate.

```
public class A
{
    public sealed override string ToString() // Virtual method inherited from class Object
    {
        return "Do not override me!";
    }
}
```

```
public class B: A
{
    public override string ToString() // Compile time error
    {
        return "An attempt to override";
    }
}
```

taglia di

Utilizzato per ottenere la dimensione in byte per un tipo non gestito

```
int byteSize = sizeof(byte) // 1
int sbyteSize = sizeof(sbyte) // 1
int shortSize = sizeof(short) // 2
int ushortSize = sizeof(ushort) // 2
int intSize = sizeof(int) // 4
int uintSize = sizeof(uint) // 4
int longSize = sizeof(long) // 8
int ulongSize = sizeof(ulong) // 8
int charSize = sizeof(char) // 2(Unicode)
int floatSize = sizeof(float) // 4
int doubleSize = sizeof(double) // 8
int decimalSize = sizeof(decimal) // 16
int boolSize = sizeof(bool) // 1
```

statico

Il modificatore `static` viene utilizzato per dichiarare un membro statico, che non ha bisogno di essere istanziato per poter accedere, ma è invece accessibile semplicemente attraverso il suo nome, ad es. `DateTime.Now` .

`static` può essere utilizzato con classi, campi, metodi, proprietà, operatori, eventi e costruttori.

Mentre un'istanza di una classe contiene una copia separata di tutti i campi di istanza della classe, c'è solo una copia di ogni campo statico.

```
class A
{
    static public int count = 0;

    public A()
    {
        count++;
    }
}

class Program
{
    static void Main(string[] args)
    {
        A a = new A();
        A b = new A();
        A c = new A();
```

```
        Console.WriteLine(A.count); // 3
    }
}
```

`count` equivale al numero totale di istanze della classe `A`

Il modificatore statico può anche essere utilizzato per dichiarare un costruttore statico per una classe, per inizializzare dati statici o eseguire codice che deve essere chiamato una sola volta. I costruttori statici vengono chiamati prima che la classe venga referenziata per la prima volta.

```
class A
{
    static public DateTime InitializationTime;

    // Static constructor
    static A()
    {
        InitializationTime = DateTime.Now;
        // Guaranteed to only run once
        Console.WriteLine(InitializationTime.ToString());
    }
}
```

Una `static class` è contrassegnata con la parola chiave `static` e può essere utilizzata come contenitore utile per un insieme di metodi che funzionano sui parametri, ma non necessariamente richiedono l'associazione con un'istanza. A causa della natura `static` della classe, non può essere istanziato, ma può contenere un `static constructor` . Alcune funzionalità di una `static class` includono:

- Non può essere ereditato
- Non può ereditare da qualcosa di diverso da `Object`
- Può contenere un costruttore statico ma non un costruttore di istanze
- Può contenere solo membri statici
- È sigillato

Il compilatore è anche amichevole e permetterà allo sviluppatore di sapere se esistono membri di istanze all'interno della classe. Un esempio potrebbe essere una classe statica che converte tra metriche statunitensi e canadesi:

```
static class ConversionHelper {
    private static double oneGallonPerLitreRate = 0.264172;

    public static double litreToGallonConversion(int litres) {
        return litres * oneGallonPerLitreRate;
    }
}
```

Quando le classi sono dichiarate statiche:

```
public static class Functions
{
```

```
  public static int Double(int value)
  {
    return value + value;
  }
}
```

anche tutte le funzioni, proprietà o membri all'interno della classe devono essere dichiarati statici. Nessuna istanza della classe può essere creata. In sostanza, una classe statica consente di creare gruppi di funzioni raggruppate in modo logico.

Dal momento che C # 6 `static` può essere utilizzato anche a fianco `using` importare soci e metodi statici. Possono essere usati quindi senza nome della classe.

Vecchio modo, senza `using static` :

```
using System;

public class ConsoleApplication
{
    public static void Main()
    {
         Console.WriteLine("Hello World!"); //Writeline is method belonging to static class
Console
    }

}
```

Esempio con `using static`

```
using static System.Console;

public class ConsoleApplication
{
    public static void Main()
    {
         WriteLine("Hello World!"); //Writeline is method belonging to static class Console
    }

}
```

svantaggi

Mentre le classi statiche possono essere incredibilmente utili, vengono fornite con i loro avvertimenti:

- Una volta che la classe statica è stata chiamata, la classe viene caricata in memoria e non può essere eseguita attraverso il garbage collector finché non viene scaricato AppDomain che ospita la classe statica.
- Una classe statica non può implementare un'interfaccia.

int

`int` è un alias per `System.Int32` , che è un tipo di dati per interi a 32 bit con segno. Questo tipo di dati può essere trovato in `mscorlib.dll` cui fa riferimento implicitamente ogni progetto C # quando vengono creati.

Intervallo: -2.147.483.648 a 2.147.483.647

```
int int1 = -10007;
var int2 = 2132012521;
```

lungo

La parola chiave **long** viene utilizzata per rappresentare interi a 64 bit con segno. È un alias per il tipo di dati `System.Int64` presente in `mscorlib.dll` , a cui viene fatto implicitamente riferimento ogni progetto C # quando vengono creati.

*Qualsiasi variabile **lunga** può essere dichiarata sia esplicitamente che implicitamente:*

```
long long1 = 9223372036854775806;  // explicit declaration, long keyword used
var long2 = -9223372036854775806L; // implicit declaration, 'L' suffix used
```

Una variabile **lunga** può contenere qualsiasi valore da -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807 e può essere utile in situazioni in cui una variabile deve contenere un valore che supera i limiti di ciò che altre variabili (come la variabile **int**) possono contenere.

ulong

Parola chiave utilizzata per numeri interi a 64 bit senza segno. Rappresenta il tipo di dati `System.UInt64` trovato in `mscorlib.dll` cui fa riferimento implicitamente ogni progetto C # quando vengono creati.

Intervallo: da 0 a 18.446.744.073.709.551.615

```
ulong veryLargeInt = 18446744073609451315;
var anotherVeryLargeInt = 15446744063609451315UL;
```

dinamico

La parola chiave `dynamic` viene utilizzata con oggetti digitati dinamicamente . Gli oggetti dichiarati come `dynamic` escludono i controlli statici in fase di compilazione e vengono invece valutati in fase di runtime.

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
```

```
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

L'esempio seguente utilizza la `dynamic` con la libreria di Newtonsoft Json.NET, per leggere facilmente i dati da un file JSON deserializzato.

```
try
{
    string json = @"{ x : 10, y : ""ho""}";
    dynamic deserializedJson = JsonConvert.DeserializeObject(json);
    int x = deserializedJson.x;
    string y = deserializedJson.y;
    // int z = deserializedJson.z; // throws RuntimeBinderException
}
catch (RuntimeBinderException e)
{
    // This exception is thrown when a property
    // that wasn't assigned to a dynamic variable is used
}
```

Esistono alcune limitazioni associate alla parola chiave dinamica. Uno di questi è l'uso di metodi di estensione. Nell'esempio seguente viene aggiunto un metodo di estensione per la stringa: `SayHello` .

```
static class StringExtensions
{
    public static string SayHello(this string s) => $"Hello {s}!";
}
```

Il primo approccio sarà chiamarlo come al solito (come per una stringa):

```
var person = "Person";
Console.WriteLine(person.SayHello());

dynamic manager = "Manager";
Console.WriteLine(manager.SayHello()); // RuntimeBinderException
```

Nessun errore di compilazione, ma a runtime si ottiene una `RuntimeBinderException` . La soluzione per questo sarà chiamare il metodo di estensione tramite la classe statica:

```
var helloManager = StringExtensions.SayHello(manager);
Console.WriteLine(helloManager);
```

virtuale, override, nuovo

virtuale e override

La parola chiave `virtual` consente a un metodo, proprietà, indicizzatore o evento di essere sovrascritto da classi derivate e presenta un comportamento polimorfico. (I membri non sono

virtuali di default in C #)

```
public class BaseClass
{
    public virtual void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}
```

Per sovrascrivere un membro, la parola chiave `override` viene utilizzata nelle classi derivate. (Nota la firma dei membri deve essere identica)

```
public class DerivedClass: BaseClass
{
    public override void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}
```

Il comportamento polimorfico dei membri virtuali significa che quando viene invocato, il membro effettivo che viene eseguito viene determinato in fase di esecuzione anziché in fase di compilazione. Il membro che sovrascrive nella classe più derivata l'oggetto particolare è un'istanza di sarà quella eseguita.

In breve, l'oggetto può essere dichiarato di tipo `BaseClass` in fase di compilazione ma se in fase di esecuzione è un'istanza di `DerivedClass` il membro sottoposto a override verrà eseguito:

```
BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

L'override di un metodo è facoltativo:

```
public class SecondDerivedClass: DerivedClass {}

var obj1 = new SecondDerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

nuovo

Poiché solo i membri definiti come `virtual` sono sovrascrivibili e polimorfici, una classe derivata che ridefinisce un membro non virtuale potrebbe portare a risultati imprevisti.

```
public class BaseClass
{
    public void Foo()
    {
```

```
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

Quando ciò accade, il membro eseguito viene sempre determinato al momento della compilazione in base al tipo dell'oggetto.

- Se l'oggetto è dichiarato di tipo `BaseClass` (anche se in fase di esecuzione è di una classe derivata) viene eseguito il metodo di `BaseClass`
- Se l'oggetto viene dichiarato di tipo `DerivedClass` quindi il metodo di `DerivedClass` viene eseguito.

Questo di solito è un incidente (quando un membro viene aggiunto al tipo base dopo che uno identico è stato aggiunto al tipo derivato) e in questi scenari viene generato un avviso del compilatore **CS0108** .

Se è stato intenzionale, la `new` parola chiave viene utilizzata per sopprimere l'avviso del compilatore (e informare gli altri sviluppatori delle tue intenzioni!). il comportamento rimane lo stesso, la `new` parola chiave sopprime solo l'avviso del compilatore.

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public new void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

L'uso dell'override *non* è facoltativo

A differenza di C ++, l'uso della parola chiave `override` *non* è facoltativo:

```
public class A
{
    public virtual void Foo()
    {
    }
}

public class B : A
{
    public void Foo() // Generates CS0108
    {
    }
}
```

L'esempio precedente causa inoltre un avviso **CS0108** , poiché `B.Foo()` non sovrascrive automaticamente `A.Foo()` . Aggiungi `override` quando l'intenzione è di ignorare la classe base e causare comportamenti polimorfici, aggiungere `new` quando si desidera un comportamento non polimorfico e risolvere la chiamata utilizzando il tipo statico. Quest'ultimo dovrebbe essere usato con cautela, in quanto potrebbe causare grave confusione.

Il seguente codice produce persino un errore:

```
public class A
{
    public void Foo()
    {
    }
}

public class B : A
{
    public override void Foo() // Error: Nothing to override
    {
    }
}
```

Le classi derivate possono introdurre il polimorfismo

Il seguente codice è perfettamente valido (anche se raro):

```
    public class A
    {
        public void Foo()
        {
            Console.WriteLine("A");
```

```
        }
    }

    public class B : A
    {
        public new virtual void Foo()
        {
            Console.WriteLine("B");
        }
    }
```

Ora tutti gli oggetti con un riferimento statico di B (e le sue derivate) usano il polimorfismo per risolvere `Foo()` , mentre i riferimenti di A usano `A.Foo()` .

```
A a = new A();
a.Foo(); // Prints "A";
a = new B();
a.Foo(); // Prints "A";
B b = new B();
b.Foo(); // Prints "B";
```

I metodi virtuali non possono essere privati

Il compilatore C # è severo nel prevenire costrutti senza senso. I metodi contrassegnati come `virtual` non possono essere privati. Poiché un metodo privato non può essere visto da un tipo derivato, non può essere sovrascritto. Questo non riesce a compilare:

```
public class A
{
    private virtual void Foo() // Error: virtual methods cannot be private
    {
    }
}
```

asincrono, attendi

La parola chiave `await` stata aggiunta come parte della release C # 5.0 supportata da Visual Studio 2012 in poi. Sfrutta la Task Parallel Library (TPL) che ha reso la multi-threading relativamente più semplice. Le parole chiave `async` e `await` sono utilizzate in coppia con la stessa funzione mostrata di seguito. La parola chiave `await` viene utilizzata per sospendere l'esecuzione del metodo asincrono corrente fino a quando l'attività asincrona attesa viene completata e / oi relativi risultati restituiti. Per utilizzare la parola chiave `await` , il metodo che lo utilizza deve essere contrassegnato con la parola chiave `async` .

L'uso `async` con `void` è fortemente scoraggiato. Per maggiori informazioni puoi guardare qui .

Esempio:

```
public async Task DoSomethingAsync()
{
```

```
    Console.WriteLine("Starting a useless process...");
    Stopwatch stopwatch = Stopwatch.StartNew();
    int delay = await UselessProcessAsync(1000);
    stopwatch.Stop();
    Console.WriteLine("A useless process took {0} milliseconds to execute.",
stopwatch.ElapsedMilliseconds);
}

public async Task<int> UselessProcessAsync(int x)
{
    await Task.Delay(x);
    return x;
}
```

Produzione:

> "Avvio di un processo inutile ..."
>
> ** ... 1 secondo di ritardo ... **
>
> "Un processo inutile ha richiesto 1000 millisecondi."

La parola chiave pairs `async` e `await` può essere omessa se un `Task` o `Task<T>` return method restituisce solo una singola operazione asincrona.

Piuttosto che questo:

```
public async Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    await Task.Delay(x);
}
```

Si preferisce fare questo:

```
public Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    return Task.Delay(x);
}
```

5.0

In C # 5.0 `await` non può essere utilizzato in `catch` e `finally` .

6.0

Con C # 6.0 è possibile `await` nel `catch` e `finally` .

carbonizzare

Un carattere è una singola lettera memorizzata all'interno di una variabile. È un tipo di valore incorporato che richiede due byte di spazio di memoria. Rappresenta il tipo di dati `System.Char`

trovato in `mscorlib.dll` cui fa riferimento implicitamente ogni progetto C # quando vengono creati.

Ci sono molti modi per farlo.

1. `char c = 'c';`
2. `char c = '\u0063'; //Unicode`
3. `char c = '\x0063'; //Hex`
4. `char c = (char)99;//Integral`

Un char può essere implicitamente convertito in `ushort, int, uint, long, ulong, float, double,` o `decimal` e restituirà il valore intero di quel char.

```
ushort u = c;
```

restituisce 99 ecc.

Tuttavia, non ci sono conversioni implicite da altri tipi a char. Invece devi castarli.

```
ushort u = 99;
 char c = (char)u;
```

serratura

`lock` fornisce thread-safety per un blocco di codice, in modo che sia accessibile da un solo thread all'interno dello stesso processo. Esempio:

```
private static object _lockObj = new object();
static void Main(string[] args)
{
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());

    Console.ReadKey();
}

private static void TaskWork()
{
    lock(_lockObj)
    {
        Console.WriteLine("Entered");

        Task.Delay(3000);
        Console.WriteLine("Done Delaying");

        // Access shared resources safely

        Console.WriteLine("Leaving");
    }
}

Output:

Entered
Done Delaying
```

```
Leaving
Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
```

Casi d'uso:

Ogni volta che si dispone di un blocco di codice che potrebbe produrre effetti collaterali se eseguito da più thread contemporaneamente. La parola chiave di blocco insieme a un **oggetto di sincronizzazione condiviso** (`_objLock` nell'esempio) può essere utilizzata per impedirlo.

Si noti che `_objLock` non può essere `null` e più thread che eseguono il codice devono utilizzare la stessa istanza di oggetto (rendendola un campo `static` o utilizzando la stessa istanza di classe per entrambi i thread)

Dal lato del compilatore, la parola chiave lock è uno zucchero sintattico che viene sostituito da `Monitor.Enter(_lockObj);` e `Monitor.Exit(_lockObj);` . Quindi se sostituisci il blocco circondando il blocco di codice con questi due metodi, otterresti gli stessi risultati. È possibile visualizzare il codice effettivo in zucchero sintattico in C # - esempio di blocco

nullo

Una variabile di un tipo di riferimento può contenere un riferimento valido a un'istanza o un riferimento null. Il riferimento null è il valore predefinito delle variabili di tipo di riferimento, nonché i tipi di valori nullable.

`null` è la parola chiave che rappresenta un riferimento null.

Come espressione, può essere usato per assegnare il riferimento null alle variabili dei tipi sopra citati:

```
object a = null;
string b = null;
int? c = null;
List<int> d  = null;
```

Ai tipi di valori non annullabili non può essere assegnato un riferimento null. Tutti i seguenti incarichi non sono validi:

```
int a = null;
float b = null;
decimal c = null;
```

Il riferimento null *non* deve essere confuso con istanze valide di vari tipi come:

- una lista vuota (`new List<int>()`)
- una stringa vuota (`""`)

- il numero zero (`0` , `0f` , `0m`)
- il carattere null (`'\0'`)

A volte, è significativo verificare se qualcosa è nullo o un oggetto vuoto / predefinito. Il metodo System.String.IsNullOrEmpty (String) può essere utilizzato per verificare questo, oppure è possibile implementare il proprio metodo equivalente.

```
private void GreetUser(string userName)
{
    if (String.IsNullOrEmpty(userName))
    {
        //The method that called us either sent in an empty string, or they sent us a null
reference. Either way, we need to report the problem.
        throw new InvalidOperationException("userName may not be null or empty.");
    }
    else
    {
        //userName is acceptable.
        Console.WriteLine("Hello, " + userName + "!");
    }
}
```

interno

La parola chiave `internal` è un modificatore di accesso per i membri di tipo e tipo. Tipi interni o membri sono **accessibili solo all'interno di file nello stesso assembly**

utilizzo:

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

La differenza tra i diversi modificatori di accesso è chiarita qui

Modificatori di accesso

pubblico

È possibile accedere al tipo o al membro tramite qualsiasi altro codice nello stesso assembly o in un altro assembly che lo faccia riferimento.

privato

È possibile accedere al tipo o al membro solo tramite codice nella stessa classe o struttura.

protetta

È possibile accedere al tipo o al membro solo tramite codice nella stessa classe o struttura o in una classe derivata.

interno

È possibile accedere al tipo o al membro tramite qualsiasi codice nello stesso assembly, ma non da un altro assembly.

protetto interno

È possibile accedere al tipo o al membro da qualsiasi codice nello stesso assembly o da qualsiasi classe derivata in un altro assembly.

Se **non** è impostato alcun modificatore di accesso, viene utilizzato un modificatore di accesso predefinito. Quindi c'è sempre qualche forma di modificatore di accesso anche se non è impostato.

dove

`where` può servire a due scopi in C #: digita il vincolo in un argomento generico e filtra le query LINQ.

In una classe generica, consideriamo

```
public class Cup<T>
{
    // ...
}
```

T è chiamato parametro di tipo. La definizione della classe può imporre vincoli sui tipi reali che possono essere forniti per T.

I seguenti tipi di vincoli possono essere applicati:

- tipo di valore
- tipo di riferimento
- costruttore predefinito
- eredità e attuazione

tipo di valore

In questo caso possono essere fornite solo `struct` s (questo include i tipi di dati 'primitivi' come `int` , `boolean` ecc.)

```
public class Cup<T> where T : struct
{
    // ...
}
```

tipo di riferimento

In questo caso possono essere forniti solo tipi di classe

```
public class Cup<T> where T : class
{
    // ...
}
```

valore ibrido / tipo di riferimento

Occasionalmente si desidera limitare gli argomenti di tipo a quelli disponibili in un database e questi di solito si associano a tipi di valore e stringhe. Poiché tutte le restrizioni di tipo devono essere soddisfatte, non è possibile specificare `where T : struct or string` (questa non è una sintassi valida). Una soluzione alternativa consiste nel limitare gli argomenti di tipo a `IConvertible` che include tipi di "... Booleano, SByte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal, DateTime, Char e String. " È possibile che altri oggetti implementino IConvertible, sebbene ciò sia raro nella pratica.

```
public class Cup<T> where T : IConvertible
{
    // ...
}
```

costruttore predefinito

Saranno consentiti solo i tipi che contengono un costruttore predefinito. Ciò include i tipi di valore e le classi che contengono un costruttore predefinito (senza parametri)

```
public class Cup<T> where T : new
{
    // ...
}
```

eredità e attuazione

Possono essere forniti solo i tipi che ereditano da una determinata classe base o implementano una determinata interfaccia.

```
public class Cup<T> where T : Beverage
{
    // ...
}


public class Cup<T> where T : IBeer
{
    // ...
}
```

Il vincolo può anche fare riferimento a un altro parametro di tipo:

```
public class Cup<T, U> where U : T
{
    // ...
}
```

È possibile specificare più vincoli per un argomento di tipo:

```
public class Cup<T> where T : class, new()
{
    // ...
}
```

Gli esempi precedenti mostrano vincoli generici su una definizione di classe, ma i vincoli possono essere utilizzati ovunque sia fornito un argomento di tipo: classi, strutture, interfacce, metodi, ecc.

`where` può anche essere una clausola LINQ. In questo caso è analogo a `WHERE` in SQL:

```
int[] nums = { 5, 2, 1, 3, 9, 8, 6, 7, 2, 0 };

var query =
    from num in nums
    where num < 5
    select num;

    foreach (var n in query)
    {
        Console.Write(n + " ");
    }
    // prints 2 1 3 2 0
```

extern

La parola chiave `extern` viene utilizzata per dichiarare metodi implementati esternamente. Può essere utilizzato insieme all'attributo DllImport per chiamare nel codice non gestito utilizzando i servizi di interoperabilità. che in questo caso arriverà con `static` modificatore `static`

Per esempio:

```
using System.Runtime.InteropServices;
public class MyClass
{
    [DllImport("User32.dll")]
    private static extern int SetForegroundWindow(IntPtr point);

    public void ActivateProcessWindow(Process p)
    {
        SetForegroundWindow(p.MainWindowHandle);
    }
}
```

Ciò utilizza il metodo SetForegroundWindow importato dalla libreria User32.dll

Questo può anche essere usato per definire un alias di assembly esterno. che ci permette di fare

riferimento a diverse versioni degli stessi componenti dal singolo assemblaggio.

Per fare riferimento a due assembly con gli stessi nomi di tipo completi, è necessario specificare un alias al prompt dei comandi, come indicato di seguito:

```
/r:GridV1=grid.dll
/r:GridV2=grid20.dll
```

Questo crea gli alias esterni GridV1 e GridV2. Per utilizzare questi alias all'interno di un programma, consultali usando la parola chiave extern. Per esempio:

```
extern alias GridV1;
extern alias GridV2;
```

bool

Parola chiave per la memorizzazione dei valori booleani `true` e `false` . bool è un alias di System.Boolean.

Il valore predefinito di un bool è falso.

```
bool b; // default value is false
b = true; // true
b = ((5 + 2) == 6); // false
```

Per un bool per consentire valori nulli deve essere inizializzato come bool ?.

Il valore predefinito di un bool? è zero.

```
bool? a // default value is null
```

quando

Il `when` una parola chiave viene aggiunta in **C # 6** e viene utilizzata per il filtraggio delle eccezioni.

Prima dell'introduzione della parola chiave `when` , si poteva avere una clausola catch per ogni tipo di eccezione; con l'aggiunta della parola chiave, ora è possibile un controllo più dettagliato.

A `when` expression è associata a un ramo `catch` e solo se la condizione `when` è `true` , la clausola `catch` verrà eseguita. È possibile avere diverse clausole di `catch` con gli stessi tipi di classi di eccezioni e diverse `when` condizioni.

```
private void CatchException(Action action)
{
    try
    {
        action.Invoke();
    }

    // exception filter
```

```
    catch (Exception ex) when (ex.Message.Contains("when"))
    {
        Console.WriteLine("Caught an exception with when");
    }

    catch (Exception ex)
    {
        Console.WriteLine("Caught an exception without when");
    }
}

private void Method1() { throw new Exception("message for exception with when"); }
private void Method2() { throw new Exception("message for general exception"); }


CatchException(Method1);
CatchException(Method2);
```

non verificato

La parola chiave `unchecked` impedisce al compilatore di verificare la presenza di overflow / underflow.

Per esempio:

```
const int ConstantMax = int.MaxValue;
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

Senza la parola chiave non `unchecked` , nessuna delle due operazioni di aggiunta verrà compilata.

Quando è utile?

Questo è utile in quanto può aiutare ad accelerare i calcoli che sicuramente non avranno overflow dal momento che il controllo dell'overflow richiede tempo, o quando un overflow / underflow è un comportamento desiderato (ad esempio, quando si genera un codice hash).

vuoto

La parola riservata `"void"` è un alias di tipo `System.Void` e ha due usi:

1. Dichiara un metodo che non ha un valore di ritorno:

```
public void DoSomething()
{
    // Do some work, don't return any value to the caller.
}
```

Un metodo con un tipo restituito di vuoto può ancora avere la parola chiave `return` nel suo corpo.

Ciò è utile quando si desidera uscire dall'esecuzione del metodo e restituire il flusso al chiamante:

```
public void DoSomething()
{
    // Do some work...

    if (condition)
        return;

    // Do some more work if the condition evaluated to false.
}
```

2. Dichiarare un puntatore a un tipo sconosciuto in un contesto non sicuro.

In un contesto non sicuro, un tipo può essere un tipo di puntatore, un tipo di valore o un tipo di riferimento. Una dichiarazione del tipo puntatore è solitamente `type* identifier` , dove il tipo è un tipo noto, ad esempio `int* myInt` , ma può anche essere `void* identifier` , in cui il tipo è sconosciuto.

Si noti che la dichiarazione di un tipo di puntatore void è sconsigliata da Microsoft.

se, se ... altro, se ... altro se

L'istruzione `if` viene utilizzata per controllare il flusso del programma. Un'istruzione `if` identifica quale istruzione eseguire in base al valore di un'espressione `Boolean` .

Per una singola istruzione, le `braces` {} sono facoltative ma consigliate.

```
int a = 4;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
// output: "a contains an even number"
```

L' `if` può anche avere una clausola `else` , che verrà eseguita nel caso in cui la condizione sia falsa:

```
int a = 5;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
else
{
     Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number"
```

Il `if` ... `else if` construct ti permette di specificare più condizioni:

```
int a = 9;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
else if(a % 3 == 0)
{
     Console.WriteLine("a contains an odd number that is a multiple of 3");
}
else
{
     Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number that is a multiple of 3"
```

Importante notare che se una condizione è soddisfatta nell'esempio precedente, il controllo salta altri test e salta alla fine di quel particolare if else construct.So, l' *ordine* dei test è importante se si sta usando if .. else if construct

Le espressioni booleane C # utilizzano la valutazione di cortocircuito . Questo è importante nei casi in cui le condizioni di valutazione possono avere effetti collaterali:

```
if (someBooleanMethodWithSideEffects() && someOtherBooleanMethodWithSideEffects()) {
  //...
}
```

Non vi è alcuna garanzia che `someOtherBooleanMethodWithSideEffects` verrà effettivamente eseguito.

È anche importante nei casi in cui condizioni precedenti assicurano che è "sicuro" valutare quelli successivi. Per esempio:

```
if (someCollection != null && someCollection.Count > 0) {
   // ..
}
```

L'ordine è molto importante in questo caso perché, se invertiamo l'ordine:

```
if (someCollection.Count > 0 && someCollection != null) {
```

getterà una `NullReferenceException` se `someCollection` è `null` .

fare

L'operatore do esegue iterazioni su un blocco di codice fino a quando una query condizionale è uguale a false. Il ciclo do-while può anche essere interrotto da un'istruzione `goto` , `return` , `break` o `throw` .

La sintassi per la parola chiave `do` è:

do { *blocco di codice;* } while (*condizione*);

Esempio:

```
int i = 0;

do
{
    Console.WriteLine("Do is on loop number {0}.", i);
} while (i++ < 5);
```

Produzione:

"Do is on loop numero 1."
"Do is on loop numero 2."
"Do is on loop numero 3."
"Do is on loop numero 4."
"Do is on loop numero 5."

A differenza del ciclo `while` , il ciclo do-while è **Exit Controlled** . Ciò significa che il ciclo do-while eseguirà le sue istruzioni almeno una volta, anche se la condizione non riesce la prima volta.

```
bool a = false;

do
{
    Console.WriteLine("This will be printed once, even if a is false.");
} while (a == true);
```

operatore

La maggior parte degli operatori integrati (inclusi gli operatori di conversione) può essere sovraccaricata utilizzando la parola chiave `operator` insieme ai modificatori `public` e `static` .

Gli operatori sono disponibili in tre forme: operatori unari, operatori binari e operatori di conversione.

Gli operatori unari e binari richiedono almeno un parametro dello stesso tipo del tipo contenente e alcuni richiedono un operatore di corrispondenza complementare.

Gli operatori di conversione devono convertire in o dal tipo di allegato.

```
public struct Vector32
{

    public Vector32(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; }
```

```
    public int Y { get; }

    public static bool operator ==(Vector32 left, Vector32 right)
        => left.X == right.X && left.Y == right.Y;

    public static bool operator !=(Vector32 left, Vector32 right)
        => !(left == right);

    public static Vector32 operator +(Vector32 left, Vector32 right)
        => new Vector32(left.X + right.X, left.Y + right.Y);

    public static Vector32 operator +(Vector32 left, int right)
        => new Vector32(left.X + right, left.Y + right);

    public static Vector32 operator +(int left, Vector32 right)
        => right + left;

    public static Vector32 operator -(Vector32 left, Vector32 right)
        => new Vector32(left.X - right.X, left.Y - right.Y);

    public static Vector32 operator -(Vector32 left, int right)
        => new Vector32(left.X - right, left.Y - right);

    public static Vector32 operator -(int left, Vector32 right)
        => right - left;

    public static implicit operator Vector64(Vector32 vector)
        => new Vector64(vector.X, vector.Y);

    public override string ToString() => $"{{{X}, {Y}}}";

}

public struct Vector64
{

    public Vector64(long x, long y)
    {
        X = x;
        Y = y;
    }

    public long X { get; }
    public long Y { get; }

    public override string ToString() => $"{{{X}, {Y}}}";

}
```

Esempio

```
var vector1 = new Vector32(15, 39);
var vector2 = new Vector32(87, 64);

Console.WriteLine(vector1 == vector2); // false
Console.WriteLine(vector1 != vector2); // true
Console.WriteLine(vector1 + vector2);  // {102, 103}
Console.WriteLine(vector1 - vector2);  // {-72, -25}
```

struct

Un tipo di `struct` è un tipo di valore che viene in genere utilizzato per incapsulare piccoli gruppi di variabili correlate, come le coordinate di un rettangolo o le caratteristiche di un articolo in un inventario.

> Le classi sono tipi di riferimento, le strutture sono tipi di valore.

```
using static System.Console;

namespace ConsoleApplication1
{
    struct Point
    {
        public int X;
        public int Y;

        public override string ToString()
        {
            return $"X = {X}, Y = {Y}";
        }

        public void Display(string name)
        {
            WriteLine(name + ": " + ToString());
        }
    }

    class Program
    {
        static void Main()
        {
            var point1 = new Point {X = 10, Y = 20};
            // it's not a reference but value type
            var point2 = point1;
            point2.X = 777;
            point2.Y = 888;
            point1.Display(nameof(point1)); // point1: X = 10, Y = 20
            point2.Display(nameof(point2)); // point2: X = 777, Y = 888

            ReadKey();
        }
    }
}
```

Le strutture possono anche contenere costruttori, costanti, campi, metodi, proprietà, indicizzatori, operatori, eventi e tipi annidati, sebbene se siano richiesti più di questi membri, dovresti prendere in considerazione la creazione di una classe invece.

Alcuni suggerimenti da MS su quando usare struct e quando usare la classe:

TENERE CONTO

definire una struct invece di una classe se le istanze del tipo sono piccole e comunemente di

breve durata o sono comunemente incorporate in altri oggetti.

EVITARE

definire una struttura a meno che il tipo abbia tutte le seguenti caratteristiche:

- Rappresenta logicamente un singolo valore, simile ai tipi primitivi (int, double, ecc.)
- Ha una dimensione dell'istanza inferiore a 16 byte.
- È immutabile.
- Non dovrà essere incassato frequentemente.

interruttore

L'istruzione `switch` è un'istruzione di controllo che seleziona una sezione switch da eseguire da un elenco di candidati. Una dichiarazione switch include una o più sezioni switch. Ogni sezione switch contiene una o più etichette `case` seguite da una o più istruzioni. Se nessuna etichetta case contiene un valore corrispondente, il controllo viene trasferito alla sezione `default` , se ce n'è uno. Caso fall-through non è supportato in C #, in senso stretto. Tuttavia, se 1 o più etichette del `case` sono vuote, l'esecuzione seguirà il codice del successivo blocco del `case` che contiene il codice. Ciò consente il raggruppamento di più `case` con la stessa implementazione. Nell'esempio seguente, se il `month` uguale a 12, il codice nel `case 2` verrà eseguito poiché le etichette del `case 12` `1` e `2` sono raggruppate. Se un blocco del `case` non è vuoto, deve essere presente `break` prima della successiva etichetta del `case` , altrimenti il compilatore segnala un errore.

```
int month = DateTime.Now.Month; // this is expected to be 1-12 for Jan-Dec

switch (month)
{
    case 12:
    case 1:
    case 2:
        Console.WriteLine("Winter");
        break;
    case 3:
    case 4:
    case 5:
        Console.WriteLine("Spring");
        break;
    case 6:
    case 7:
    case 8:
        Console.WriteLine("Summer");
        break;
    case 9:
    case 10:
    case 11:
        Console.WriteLine("Autumn");
        break;
    default:
        Console.WriteLine("Incorrect month index");
        break;
}
```

Un `case` può essere etichettato solo da un valore noto al *momento della compilazione* (es. `1` , `"str"`

, `Enum.A`), quindi una `variable` non è un'etichetta `case` valida, ma un valore `const` o un valore `Enum` è (così come qualsiasi valore letterale).

interfaccia

Un `interface` contiene le firme di metodi, proprietà ed eventi. Le classi derivate definiscono i membri poiché l'interfaccia contiene solo la dichiarazione dei membri.

Un'interfaccia è dichiarata usando la parola chiave `interface` .

```
interface IProduct
{
    decimal Price { get; }
}

class Product : IProduct
{
    const decimal vat = 0.2M;

    public Product(decimal price)
    {
        _price = price;
    }

    private decimal _price;
    public decimal Price { get { return _price * (1 + vat); } }
}
```

pericoloso

La parola chiave `unsafe` può essere utilizzata nelle dichiarazioni del tipo o del metodo o per dichiarare un blocco in linea.

Lo scopo di questa parola chiave è di abilitare l'uso del *sottoinsieme non sicuro* di C # per il blocco in questione. Il sottoinsieme non sicuro include funzionalità come puntatori, allocazione dello stack, array di tipo C e così via.

Il codice non sicuro non è verificabile ed è per questo che il suo utilizzo è scoraggiato. La compilazione di codice non sicuro richiede il passaggio di un passaggio al compilatore C #. Inoltre, il CLR richiede che l'assembly in esecuzione abbia piena fiducia.

Nonostante queste limitazioni, il codice non sicuro ha degli usi validi per rendere alcune operazioni più performanti (ad esempio l'indicizzazione degli array) o più semplici (ad es. Interop con alcune librerie non gestite).

Come un esempio molto semplice

```
// compile with /unsafe
class UnsafeTest
{
   unsafe static void SquarePtrParam(int* p)
   {
      *p *= *p; // the '*' dereferences the pointer.
```

```
      //Since we passed in "the address of i", this becomes "i *= i"
   }

   unsafe static void Main()
   {
      int i = 5;
      // Unsafe method: uses address-of operator (&):
      SquarePtrParam(&i); // "&i" means "the address of i". The behavior is similar to "ref i"
      Console.WriteLine(i); // Output: 25
   }
}
```

Mentre lavoriamo con i puntatori, possiamo modificare direttamente i valori delle locazioni di memoria, piuttosto che doverli affrontare per nome. Si noti che questo spesso richiede l'uso della parola chiave fissa per prevenire possibili corruzioni della memoria in quanto il garbage collector sposta le cose intorno (altrimenti, si potrebbe ottenere l' errore CS0212). Poiché non è possibile scrivere su una variabile che è stata "riparata", spesso è necessario avere un secondo puntatore che inizi a puntare alla stessa posizione del primo.

```
void Main()
{
    int[] intArray = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    UnsafeSquareArray(intArray);
    foreach(int i in intArray)
        Console.WriteLine(i);
}

unsafe static void UnsafeSquareArray(int[] pArr)
{
    int len = pArr.Length;

    //in C or C++, we could say
    // int* a = &(pArr[0])
    // however, C# requires you to "fix" the variable first
    fixed(int* fixedPointer = &(pArr[0]))
    {
        //Declare a new int pointer because "fixedPointer" cannot be written to.
        // "p" points to the same address space, but we can modify it
        int* p = fixedPointer;

        for (int i = 0; i < len; i++)
        {
            *p *= *p; //square the value, just like we did in SquarePtrParam, above
            p++;      //move the pointer to the next memory space.
                      // NOTE that the pointer will move 4 bytes since "p" is an
                      // int pointer and an int takes 4 bytes

            //the above 2 lines could be written as one, like this:
            // "*p *= *p++;"
        }
    }
}
```

Produzione:

```
1
```

```
4
9
16
25
36
49
64
81
100
```

`unsafe` consente inoltre l'uso di stackalloc che allocherà memoria nello stack come _alloca nella libreria di runtime C. Possiamo modificare l'esempio precedente per usare `stackalloc` come segue:

```
unsafe void Main()
{
    const int len=10;
    int* seedArray = stackalloc int[len];

    //We can no longer use the initializer "{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}" as before.
    // We have at least 2 options to populate the array. The end result of either
    // option will be the same (doing both will also be the same here).

    //FIRST OPTION:
    int* p = seedArray; // we don't want to lose where the array starts, so we
                        // create a shadow copy of the pointer
    for(int i=1; i<=len; i++)
        *p++ = i;
    //end of first option

    //SECOND OPTION:
    for(int i=0; i<len; i++)
        seedArray[i] = i+1;
    //end of second option

    UnsafeSquareArray(seedArray, len);
    for(int i=0; i< len; i++)
        Console.WriteLine(seedArray[i]);
}

//Now that we are dealing directly in pointers, we don't need to mess around with
// "fixed", which dramatically simplifies the code
unsafe static void UnsafeSquareArray(int* p, int len)
{
    for (int i = 0; i < len; i++)
        *p *= *p++;
}
```

(L'uscita è la stessa di sopra)

implicito

La parola chiave `implicit` viene utilizzata per sovraccaricare un operatore di conversione. Ad esempio, puoi dichiarare una classe `Fraction` che dovrebbe essere automaticamente convertita in `double` quando necessario e che può essere convertita automaticamente da `int` :

```
class Fraction(int numerator, int denominator)
```

```
{
    public int Numerator { get; } = numerator;
    public int Denominator { get; } = denominator;
    // ...
    public static implicit operator double(Fraction f)
    {
        return f.Numerator / (double) f.Denominator;
    }
    public static implicit operator Fraction(int i)
    {
        return new Fraction(i, 1);
    }
}
```

vero falso

Le parole chiave `true` e `false` hanno due usi:

1. Come valori booleani letterali

```
var myTrueBool = true;
var myFalseBool = false;
```

2. Come operatori che possono essere sovraccaricati

```
public static bool operator true(MyClass x)
{
    return x.value >= 0;
}

public static bool operator false(MyClass x)
{
    return x.value < 0;
}
```

Il sovraccarico dell'operatore falso era utile prima del C # 2.0, prima dell'introduzione dei tipi `Nullable` .
Un tipo che sovraccarica l'operatore `true` , deve anche sovraccaricare l'operatore `false` .

stringa

`string` è un alias del tipo di dati .NET `System.String` , che consente di memorizzare testo (sequenze di caratteri).

Notazione:

```
string a = "Hello";
var b = "world";
var f = new string(new []{ 'h', 'i', '!' }); // hi!
```

Ogni carattere nella stringa è codificato in UTF-16, il che significa che ogni carattere richiederà almeno 2 byte di spazio di archiviazione.

USHORT

Un tipo numerico utilizzato per memorizzare interi positivi a 16 bit. `ushort` è un alias per `System.UInt16` e occupa 2 byte di memoria.

L'intervallo valido è compreso tra `0` e `65535` .

```
ushort a = 50; // 50
ushort b = 65536; // Error, cannot be converted
ushort c = unchecked((ushort)65536); // Overflows (wraps around to 0)
```

sbyte

Un tipo numerico utilizzato per memorizzare interi con *segno* a 8 bit. `sbyte` è un alias per `System.SByte` e occupa 1 byte di memoria. Per l'equivalente senza segno, usa il `byte` .

L'intervallo valido è compreso tra `-127` e `127` (l'avanzo viene utilizzato per memorizzare il segno).

```
sbyte a = 127; // 127
sbyte b = -127; // -127
sbyte c = 200; // Error, cannot be converted
sbyte d = unchecked((sbyte)129); // -127 (overflows)
```

var

Una variabile locale implicitamente tipizzata che è fortemente tipizzata come se l'utente avesse dichiarato il tipo. A differenza di altre dichiarazioni variabili, il compilatore determina il tipo di variabile che rappresenta in base al valore che gli viene assegnato.

```
var i = 10; // implicitly typed, the compiler must determine what type of variable this is
int i = 10; // explicitly typed, the type of variable is explicitly stated to the compiler

// Note that these both represent the same type of variable (int) with the same value (10).
```

A differenza di altri tipi di variabili, le definizioni di variabili con questa parola chiave devono essere inizializzate quando dichiarate. Ciò è dovuto alla parola chiave **var che** rappresenta una variabile tipizzata implicitamente.

```
var i;
i = 10;

// This code will not run as it is not initialized upon declaration.
```

La parola chiave **var** può anche essere utilizzata per creare nuovi tipi di dati al volo. Questi nuovi tipi di dati sono noti come *tipi anonimi* . Sono abbastanza utili, in quanto consentono a un utente di definire un insieme di proprietà senza dover dichiarare esplicitamente qualsiasi tipo di tipo di oggetto per primo.

Semplice tipo anonimo

```
var a = new { number = 1, text = "hi" };
```

Query LINQ che restituisce un tipo anonimo

```
public class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class DogWithBreed
{
    public Dog Dog { get; set; }
    public string BreedName  { get; set; }
}

public void GetDogsWithBreedNames()
{
    var db = new DogDataContext(ConnectString);
    var result = from d in db.Dogs
              join b in db.Breeds on d.BreedId equals b.BreedId
              select new
                     {
                         DogName = d.Name,
                         BreedName = b.BreedName
                     };

    DoStuff(result);
}
```

È possibile utilizzare la parola chiave var nell'istruzione foreach

```
public bool hasItemInList(List<String> list, string stringToSearch)
{
    foreach(var item in list)
    {
        if( ( (string)item ).equals(stringToSearch) )
            return true;
    }

    return false;
}
```

delegare

I delegati sono tipi che rappresentano un riferimento a un metodo. Sono usati per passare metodi come argomenti ad altri metodi.

I delegati possono contenere metodi statici, metodi di istanza, metodi anonimi o espressioni lambda.

```
class DelegateExample
{
    public void Run()
    {
```

```
        //using class method
        InvokeDelegate( WriteToConsole );

        //using anonymous method
        DelegateInvoker di = delegate ( string input )
        {
            Console.WriteLine( string.Format( "di: {0} ", input ) );
            return true;
        };
        InvokeDelegate( di );

        //using lambda expression
        InvokeDelegate( input => false );
    }

    public delegate bool DelegateInvoker( string input );

    public void InvokeDelegate(DelegateInvoker func)
    {
        var ret = func( "hello world" );
        Console.WriteLine( string.Format( " > delegate returned {0}", ret ) );
    }

    public bool WriteToConsole( string input )
    {
        Console.WriteLine( string.Format( "WriteToConsole: '{0}'", input ) );
        return true;
    }
}
```

Quando si assegna un metodo a un delegato è importante notare che il metodo deve avere lo stesso tipo di ritorno e anche i parametri. Ciò differisce dal sovraccarico del metodo "normale", in cui solo i parametri definiscono la firma del metodo.

Gli eventi sono costruiti in cima ai delegati.

evento

Un `event` consente allo sviluppatore di implementare un modello di notifica.

Semplice esempio

```
public class Server
{
    // defines the event
    public event EventHandler DataChangeEvent;

    void RaiseEvent()
    {
        var ev = DataChangeEvent;
        if(ev != null)
        {
            ev(this, EventArgs.Empty);
        }
    }
}
```

```
public class Client
{
    public void Client(Server server)
    {
        // client subscribes to the server's DataChangeEvent
        server.DataChangeEvent += server_DataChanged;
    }

    private void server_DataChanged(object sender, EventArgs args)
    {
        // notified when the server raises the DataChangeEvent
    }
}
```

Riferimento MSDN

parziale

La parola chiave `partial` può essere utilizzata durante la definizione del tipo di classe, struct o interfaccia per consentire la divisione della definizione del tipo in più file. Questo è utile per incorporare nuove funzionalità nel codice generato automaticamente.

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
    }
}
```

Nota: una classe può essere suddivisa in un numero qualsiasi di file. Tuttavia, tutte le dichiarazioni devono trovarsi nello stesso spazio dei nomi e nello stesso assembly.

I metodi possono anche essere dichiarati parziali usando la parola chiave `partial` . In questo caso un file conterrà solo la definizione del metodo e un altro file conterrà l'implementazione.

> Un metodo parziale ha la propria firma definita in una parte di un tipo parziale e la sua implementazione definita in un'altra parte del tipo. I metodi parziali consentono ai progettisti di classi di fornire hook di metodo, simili ai gestori di eventi, che gli sviluppatori possono decidere di implementare o meno. Se lo sviluppatore non fornisce un'implementazione, il compilatore rimuove la firma al momento della compilazione. Le

seguenti condizioni si applicano ai metodi parziali:

- Le firme in entrambe le parti del tipo parziale devono corrispondere.
- Il metodo deve restituire void.
- Non sono ammessi modificatori di accesso. I metodi parziali sono implicitamente privati.

- MSDN

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
        public partial Method1(string str);
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
        public partial Method1(string str)
        {
            Console.WriteLine(str);
        }
    }
}
```

Nota: anche il tipo contenente il metodo parziale deve essere dichiarato parziale.

Leggi parole online: https://riptutorial.com/it/csharp/topic/26/parole

Capitolo 121: Per iniziare: Json con C

introduzione

Il seguente argomento introdurrà un modo per lavorare con Json usando il linguaggio C # e concetti di serializzazione e deserializzazione.

Examples

Semplice esempio di JSON

```
{
    "id": 89,
    "name": "Aldous Huxley",
    "type": "Author",
    "books":[{
                "name": "Brave New World",
                "date": 1932
             },
             {
                "name": "Eyeless in Gaza",
                "date": 1936
             },
             {
                "name": "The Genius and the Goddess",
                "date": 1955
             }]
}
```

Se sei nuovo in Json, ecco un tutorial esemplificato .

Per prima cosa: Libreria per lavorare con Json

Per lavorare con Json usando C #, è necessario utilizzare Newtonsoft (.net library). Questa libreria fornisce metodi che consentono al programmatore di serializzare e deserializzare oggetti e altro ancora. C'è un tutorial se vuoi sapere i dettagli sui suoi metodi e usi.

Se utilizzi Visual Studio, vai su *Strumenti / Gestore pacchetti Nuget / Gestisci pacchetto su soluzione* / e digita "Newtonsoft" nella barra di ricerca e installa il pacchetto. Se non hai NuGet, questo tutorial dettagliato potrebbe aiutarti.

Implementazione C

Prima di leggere del codice, è importante comprendere i concetti principali che aiuteranno a programmare le applicazioni usando json.

> **Serializzazione** : processo di conversione di un oggetto in un flusso di byte che può essere inviato attraverso le applicazioni. Il seguente codice può essere serializzato e

convertito nel json precedente.

Deserializzazione : processo di conversione di un json / flusso di byte in un oggetto. È esattamente il processo opposto di serializzazione. Il json precedente può essere deserializzato in un oggetto C # come dimostrato negli esempi di seguito.

Per risolvere questo problema, è importante trasformare la struttura di JSON in classi per utilizzare i processi già descritti. Se si utilizza Visual Studio, è possibile trasformare automaticamente un json in una classe semplicemente selezionando *"Modifica / Incolla speciale / Incolla JSON come classi"* e incollando la struttura di JSON.

```
using Newtonsoft.Json;

  class Author
{
    [JsonProperty("id")] // Set the variable below to represent the json attribute
    public int id;       //"id"
    [JsonProperty("name")]
    public string name;
    [JsonProperty("type")]
    public string type;
    [JsonProperty("books")]
    public Book[] books;

    public Author(int id, string name, string type, Book[] books) {
        this.id = id;
        this.name = name;
        this.type= type;
        this.books = books;
    }
}

 class Book
{
   [JsonProperty("name")]
   public string name;
   [JsonProperty("date")]
   public DateTime date;
}
```

serializzazione

```
 static void Main(string[] args)
    {
        Book[] books = new Book[3];
        Author author = new Author(89,"Aldous Huxley","Author",books);
        string objectDeserialized = JsonConvert.SerializeObject(author);
        //Converting author into json
    }
```

Il metodo ".SerializeObject" riceve come parametro un *oggetto tipo* , quindi puoi inserire qualcosa.

deserializzazione

È possibile ricevere un json da qualsiasi luogo, un file o anche un server, in modo che non sia

incluso nel seguente codice.

```
static void Main(string[] args)
{
    string jsonExample; // Has the previous json
    Author author = JsonConvert.DeserializeObject<Author>(jsonExample);
}
```

Il metodo ".DeserializeObject" deserializza " *jsonExample* " in un oggetto " *Autore* ". Questo è il motivo per cui è importante impostare le variabili json nella definizione delle classi, in modo che il metodo acceda per riempirlo.

Funzione di utilità comune di serializzazione e decodificazione

Questo esempio ha utilizzato la funzione comune per tutti i tipi di serializzazione e deserializzazione degli oggetti.

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml.Serialization;

namespace Framework
{
public static class IGUtilities
{
        public static string Serialization(this T obj)
        {
        string data = JsonConvert.SerializeObject(obj);
        return data;
        }

        public static T Deserialization(this string JsonData)
        {
        T copy = JsonConvert.DeserializeObject(JsonData);
        return copy;
        }

         public static T Clone(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            T copy = JsonConvert.DeserializeObject(data);
            return copy;
        }
}
}
```

Leggi Per iniziare: Json con C # online: https://riptutorial.com/it/csharp/topic/9910/per-iniziare--json-con-c-sharp

Capitolo 122: Polimorfismo

Examples

Un altro esempio di polimorfismo

Il polimorfismo è uno dei pilastri dell'OOP. Poly deriva da un termine greco che significa "forme multiple".

Di seguito è riportato un esempio che mostra Polymorphism. La classe `Vehicle` assume più forme come classe base.

Le classi derivate `Ducati` e `Lamborghini` ereditano da `Vehicle` e sovrascrivono il metodo `Display()` della classe base, per visualizzare il proprio `NumberOfWheels` .

```
public class Vehicle
{
    protected int NumberOfWheels { get; set; } = 0;
    public Vehicle()
    {
    }

    public virtual void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Vehicle)} is
{NumberOfWheels}");
    }
}

public class Ducati : Vehicle
{
    public Ducati()
    {
        NoOfWheels = 2;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Ducati)} is
{NumberOfWheels}");
    }
}

public class Lamborghini : Vehicle
{
    public Lamborghini()
    {
        NoOfWheels = 4;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Lamborghini)} is
{NumberOfWheels}");
    }
```

```
}
```

Di seguito è riportato lo snippet di codice in cui viene visualizzato Polymorphism. L'oggetto viene creato per il tipo di base `Vehicle` utilizza un `vehicle` variabile sulla linea 1. Chiama il metodo della classe base `Display()` sulla linea 2 e visualizza l'output come mostrato.

```
static void Main(string[] args)
{
   Vehicle vehicle = new Vehicle();     //Line 1
   vehicle.Display();                   //Line 2
   vehicle = new Ducati();              //Line 3
   vehicle.Display();                   //Line 4
   vehicle = new Lamborghini();         //Line 5
   vehicle.Display();                   //Line 6
}
```

Alla riga 3, l'oggetto `vehicle` è indirizzato alla classe derivata `Ducati` e chiama il suo metodo `Display()` , che visualizza l'uscita come mostrato. Qui viene il polimorfismo, anche se l'oggetto `vehicle` è di tipo `Vehicle` , si chiama il metodo classe derivata `Display()` come tipo `Ducati` eredita dalla classe base `Display()` metodo, poiché il `vehicle` oggetto è puntato verso `Ducati` .

La stessa spiegazione è applicabile quando richiama il metodo `Display()` del tipo `Lamborghini` .

L'uscita è mostrata sotto

```
The number of wheels for the Vehicle is 0        // Line 2
The number of wheels for the Ducati is 2         // Line 4
The number of wheels for the Lamborghini is 4    // Line 6
```

Tipi di polimorfismo

Il polimorfismo indica che un'operazione può essere applicata anche a valori di altri tipi.

Esistono diversi tipi di polimorfismo:

- **Polimorfismo ad hoc:**
  contiene `function overloading` . L'obiettivo è che un metodo può essere utilizzato con diversi tipi senza la necessità di essere generici.
- **Polimorfismo parametrico:**
  è l'uso di tipi generici. Vedi Generics
- **subtyping:**
  ha il target ereditato da una classe per generalizzare una funzionalità simile

---

Polimorfismo ad hoc

L'obiettivo del `Ad hoc polymorphism` è quello di creare un metodo, che può essere chiamato da diversi tipi di dati senza bisogno di conversione di tipo nella chiamata di funzione o generici. Il seguente metodo (s) `sumInt(par1, par2)` può essere chiamato con diversi tipi di dati e ha per ogni

combinazione di tipi una propria implementazione:

```
public static int sumInt( int a, int b)
{
    return a + b;
}

public static int sumInt( string a, string b)
{
    int _a, _b;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    if(!Int32.TryParse(b, out _b))
        _b = 0;

    return _a + _b;
}

public static int sumInt(string a, int b)
{
    int _a;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    return _a + b;
}

public static int sumInt(int a, string b)
{
    return sumInt(b,a);
}
```

Ecco una chiamata di esempio:

```
public static void Main()
{
    Console.WriteLine(sumInt( 1 , 2 ));  //  3
    Console.WriteLine(sumInt("3","4"));  //  7
    Console.WriteLine(sumInt("5", 6 ));  // 11
    Console.WriteLine(sumInt( 7 ,"8"));  // 15
}
```

---

subtyping

Sottotipo è l'uso di ereditare da una classe base per generalizzare un comportamento simile:

```
public interface Car{
    void refuel();
}

public class NormalCar : Car
{
```

```
    public void refuel()
    {
        Console.WriteLine("Refueling with petrol");
    }
}

public class ElectricCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Charging battery");
    }
}
```

Entrambe le classi `NormalCar` e `ElectricCar` ora hanno un metodo per fare rifornimento, ma la loro stessa implementazione. Ecco un esempio:

```
public static void Main()
{
    List<Car> cars = new List<Car>(){
        new NormalCar(),
        new ElectricCar()
    };

    cars.ForEach(x => x.refuel());
}
```

L'output sarà seguito:

> Rifornimento di carburante con benzina
> Carica della batteria

Leggi Polimorfismo online: https://riptutorial.com/it/csharp/topic/1589/polimorfismo

Capitolo 123: Presa asincrona

introduzione

Utilizzando socket asincroni, un server può ascoltare le connessioni in entrata e fare qualche altra logica nel frattempo in contrasto con il socket sincrono quando sono in ascolto bloccano il thread principale e l'applicazione sta diventando non rispondente e si bloccherà fino a quando un client non si connetterà.

Osservazioni

Presa e rete

Come accedere a un server al di fuori della mia rete? Questa è una domanda comune e quando viene posta la domanda è per lo più segnalata come argomento.

Lato server

Sulla rete del tuo server devi portare il tuo router in avanti al tuo server.

Per esempio PC su cui è in esecuzione il server:

IP locale = `192.168.1.115`

Il server sta ascoltando la porta 1234.

Inoltra connessioni in entrata sul router `Port 1234` a `192.168.1.115`

Dalla parte del cliente

L'unica cosa che devi cambiare è l'IP. Non vuoi collegarti al tuo indirizzo di loopback ma all'IP pubblico dalla rete su cui è in esecuzione il tuo server. Questo IP puoi arrivare qui .

```
_connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("10.10.10.10"), 1234));
```

Quindi ora crei una richiesta su questo endpoint: `10.10.10.10:1234` se hai fatto la proprietà port forwarding del router il tuo server e client si connetteranno senza alcun problema.

Se vuoi connetterti ad un IP locale non dovrai portforwart cambiare semplicemente l'indirizzo di loopback a `192.168.1.178` o qualcosa del genere.

Invio di dati:

I dati vengono inviati nell'array di byte. Devi mettere i tuoi dati in un array di byte e decomprimerlo dall'altro lato.

Se hai familiarità con il socket puoi anche provare a crittografare il tuo array di byte prima di

inviarlo. Ciò impedirà a chiunque di rubare il pacchetto.

Examples

Esempio di socket asincrono (client / server).

Esempio lato server

Crea listener per server

Inizia con la creazione di un server che gestirà i client che si connettono e le richieste che verranno inviate. Quindi crea una classe listener che gestirà questo.

```
class Listener
{
    public Socket ListenerSocket; //This is the socket that will listen to any incoming
connections
    public short Port = 1234; // on this port we will listen

    public Listener()
    {
        ListenerSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
    }
 }
```

Per prima cosa dobbiamo inizializzare il socket Listener dove possiamo ascoltare tutte le connessioni. Useremo un socket Tcp per questo usiamo SocketType.Stream. Inoltre, specifichiamo la porta strega che il server dovrebbe ascoltare

Quindi iniziamo ad ascoltare le connessioni in arrivo.

I metodi dell'albero che usiamo qui sono:

1. ListenerSocket.Bind ();

   Questo metodo associa il socket a un IPEndPoint . Questa classe contiene le informazioni sull'host e sulla porta locale o remota necessarie a un'applicazione per connettersi a un servizio su un host.

2. ListenerSocket.Listen (10);

   Il parametro backlog specifica il numero di connessioni in ingresso che possono essere accodate per l'accettazione.

3. ListenerSocket.BeginAccept ();

   Il server inizierà ad ascoltare le connessioni in entrata e continuerà con altra logica. Quando c'è una connessione il server torna a questo metodo ed eseguirà il metodo AcceptCallBack

```
    public void StartListening()
```

```
    {
        try
        {
                MessageBox.Show($"Listening started port:{Port} protocol type:
{ProtocolType.Tcp}");
                ListenerSocket.Bind(new IPEndPoint(IPAddress.Any, Port));
                ListenerSocket.Listen(10);
                ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
        }
        catch(Exception ex)
        {
            throw new Exception("listening error" + ex);
        }
    }
```

Quindi, quando un client si connette, possiamo accettarli con questo metodo:

I tre metodi che utilizziamo qui sono:

1. ListenerSocket.EndAccept ()

   Abbiamo iniziato la funzione di callback con `Listener.BeginAccept()` ora dobbiamo terminare la richiamata. `The EndAccept()` metodo `The EndAccept()` accetta un parametro IAsyncResult, questo memorizzerà lo stato del metodo asincrono, Da questo stato possiamo estrarre il socket da cui proveniva la connessione in ingresso.

2. `ClientController.AddClient()`

   Con il socket ottenuto da `EndAccept()` creiamo un Client con un metodo personalizzato *(codice ClientController sotto l'esempio del server)* .

3. ListenerSocket.BeginAccept ()

   Abbiamo bisogno di ricominciare ad ascoltare quando il socket ha finito di gestire la nuova connessione. Passa nel metodo che catturerà questa richiamata. E anche passare il socket Listener in modo da poter riutilizzare questo socket per le prossime connessioni.

```
    public void AcceptCallback(IAsyncResult ar)
    {
        try
        {
            Console.WriteLine($"Accept CallBack port:{Port} protocol type:
{ProtocolType.Tcp}");
            Socket acceptedSocket = ListenerSocket.EndAccept(ar);
            ClientController.AddClient(acceptedSocket);

            ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
        }
        catch (Exception ex)
        {
            throw new Exception("Base Accept error"+ ex);
        }
    }
```

Ora abbiamo un socket di ascolto ma come riceviamo i dati inviati dal client che è ciò che mostra il

prossimo codice.

Crea ricevitore del server per ogni cliente

Prima di creare una classe di ricezione con un costruttore che accetta un Socket come parametro:

```
    public class ReceivePacket
    {
        private byte[] _buffer;
        private Socket _receiveSocket;

        public ReceivePacket(Socket receiveSocket)
        {
           _receiveSocket = receiveSocket;
        }
    }
```

Nel prossimo metodo iniziamo con il dare al buffer una dimensione di 4 byte (Int32) o il pacchetto contiene parti {lunghezza, dati reali}. Quindi i primi 4 byte riserviamo per la lunghezza dei dati il resto per i dati effettivi.

Successivamente usiamo il metodo BeginReceive () . Questo metodo viene utilizzato per iniziare a ricevere dai client connessi e quando riceverà i dati eseguirà la funzione `ReceiveCallback` .

```
    public void StartReceiving()
    {
        try
        {
            _buffer = new byte[4];
            _receiveSocket.BeginReceive(_buffer, 0, _buffer.Length, SocketFlags.None,
ReceiveCallback, null);
        }
        catch {}
    }

    private void ReceiveCallback(IAsyncResult AR)
    {
        try
        {
            // if bytes are less than 1 takes place when a client disconnect from the server.
            // So we run the Disconnect function on the current client
            if (_receiveSocket.EndReceive(AR) > 1)
            {
                // Convert the first 4 bytes (int 32) that we received and convert it to an
Int32 (this is the size for the coming data).
                _buffer = new byte[BitConverter.ToInt32(_buffer, 0)];
                // Next receive this data into the buffer with size that we did receive before
                _receiveSocket.Receive(_buffer, _buffer.Length, SocketFlags.None);
                // When we received everything its onto you to convert it into the data that
you've send.
                // For example string, int etc... in this example I only use the
implementation for sending and receiving a string.

                // Convert the bytes to string and output it in a message box
                string data = Encoding.Default.GetString(_buffer);
                MessageBox.Show(data);
                // Now we have to start all over again with waiting for a data to come from
```

```
the socket.
                StartReceiving();
            }
            else
            {
                Disconnect();
            }
        }
        catch
        {
            // if exeption is throw check if socket is connected because than you can
startreive again else Dissconect
            if (!_receiveSocket.Connected)
            {
                Disconnect();
            }
            else
            {
                StartReceiving();
            }
        }
    }

    private void Disconnect()
    {
        // Close connection
        _receiveSocket.Disconnect(true);
        // Next line only apply for the server side receive
        ClientController.RemoveClient(_clientId);
        // Next line only apply on the Client Side receive
        Here you want to run the method TryToConnect()
    }
```

Quindi abbiamo configurato un server in grado di ricevere e ascoltare le connessioni in entrata. Quando un client lo connette, verrà aggiunto a un elenco di client e ogni client avrà la sua classe di ricezione. Per fare in modo che il server ascolti:

```
Listener listener = new Listener();
listener.StartListening();
```

Alcune classi che uso in questo esempio

```
    class Client
    {
        public Socket _socket { get; set; }
        public ReceivePacket Receive { get; set; }
        public int Id { get; set; }

        public Client(Socket socket, int id)
        {
            Receive = new ReceivePacket(socket, id);
            Receive.StartReceiving();
            _socket = socket;
            Id = id;
        }
    }

     static class ClientController
```

```
    {
        public static List<Client> Clients = new List<Client>();

        public static void AddClient(Socket socket)
        {
            Clients.Add(new Client(socket,Clients.Count));
        }

        public static void RemoveClient(int id)
        {
            Clients.RemoveAt(Clients.FindIndex(x => x.Id == id));
        }
     }
```

Esempio lato client

Connessione al server

Prima di tutto vogliamo creare una classe che collega al server il nome che gli diamo è: Connettore:

```
class Connector
{
    private Socket _connectingSocket;
}
```

Il prossimo metodo per questa classe è TryToConnect ()

Questo metodo fa alcune cose interminabili:

1. Crea il socket;
2. Prossimo ciclo I fino al collegamento della presa
3. Ogni ciclo è solo tenendo premuto il thread per 1 secondo, non vogliamo DOS sul server XD
4. Con Connect () proverà a connettersi al server. Se fallisce, genererà un'eccezione ma il wile manterrà il programma connesso al server. È possibile utilizzare un metodo Connect CallBack per questo, ma andrò a chiamare un metodo quando lo Socket è connesso.
5. Si noti che il client sta tentando di connettersi al PC locale sulla porta 1234.

```
 public void TryToConnect()
 {
     _connectingSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

      while (!_connectingSocket.Connected)
      {
          Thread.Sleep(1000);

          try
          {
              _connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("127.0.0.1"),
1234));
```

```
            }
            catch { }
        }
        SetupForReceiveing();
       }
    }

    private void SetupForReceiveing()
    {
       // View Client Class bottom of Client Example
        Client.SetClient(_connectingSocket);
        Client.StartReceiving();
    }
```

Invio di un messaggio al server

Quindi ora abbiamo quasi una finitura o l'applicazione Socket. L'unica cosa che non abbiamo jet è una classe per inviare un messaggio al server.

```
public class SendPacket
{
    private Socket _sendSocked;

    public SendPacket(Socket sendSocket)
    {
        _sendSocked = sendSocket;
    }

    public void Send(string data)
    {
        try
        {
            /* what hapends here:
                 1. Create a list of bytes
                 2. Add the length of the string to the list.
                    So if this message arrives at the server we can easily read the length of
the coming message.
                 3. Add the message(string) bytes
            */

            var fullPacket = new List<byte>();
            fullPacket.AddRange(BitConverter.GetBytes(data.Length));
            fullPacket.AddRange(Encoding.Default.GetBytes(data));

            /* Send the message to the server we are currently connected to.
            Or package stucture is {length of data 4 bytes (int32), actual data}*/
            _sendSocked.Send(fullPacket.ToArray());
        }
        catch (Exception ex)
        {
            throw new Exception();
        }
    }
```

Finaly cassa due pulsanti uno per la connessione e l'altro per l'invio di un messaggio:

```
    private void ConnectClick(object sender, EventArgs e)
    {
```

```
        Connector tpp = new Connector();
        tpp.TryToConnect();
    }

    private void SendClick(object sender, EventArgs e)
    {
        Client.SendString("Test data from client");
    }
```

La classe client che ho usato in questo esempio

```
    public static void SetClient(Socket socket)
    {
        Id = 1;
        Socket = socket;
        Receive = new ReceivePacket(socket, Id);
        SendPacket = new SendPacket(socket);
    }
```

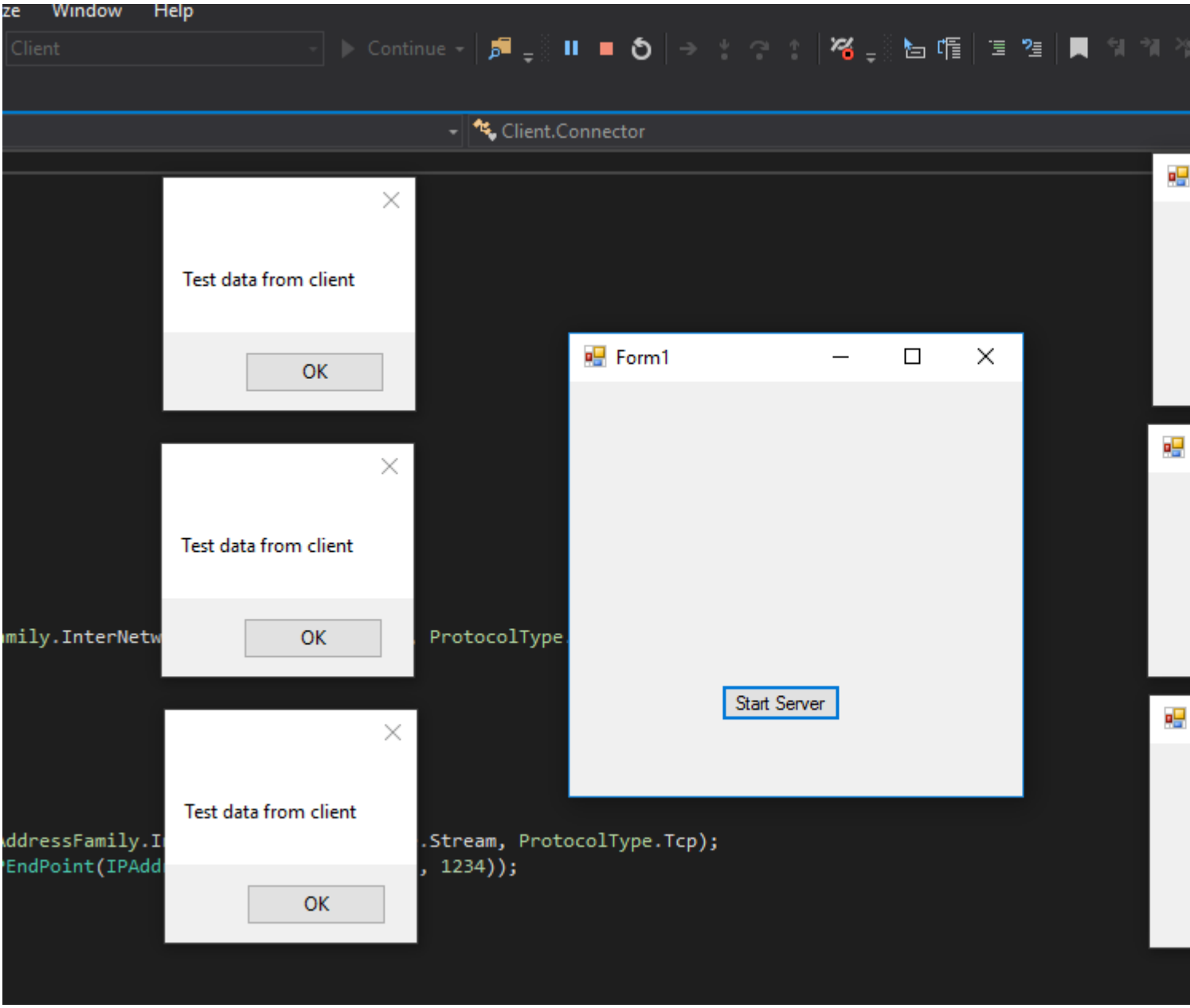
Avviso

La classe di ricezione dal server è la stessa della classe di ricezione del client.

Conclusione

Ora hai un server e un client. Puoi lavorare su questo esempio di base. Ad esempio, assicurati che il server possa anche ricevere file o altri elementi. O inviare un messaggio al cliente. Nel server hai una lista di clienti così quando ricevi qualcosa che saprai dal cliente da cui proviene.

Risultato finale:



Leggi Presa asincrona online: https://riptutorial.com/it/csharp/topic/9638/presa-asincrona

Capitolo 124: Programmazione funzionale

Examples

Func e azione

Func fornisce un supporto per le funzioni anonime parametrizzate. I tipi principali sono gli input e l'ultimo tipo è sempre il valore restituito.

```
// square a number.
Func<double, double> square = (x) => { return x * x; };

// get the square root.
// note how the signature matches the built in method.
Func<double, double> squareroot = Math.Sqrt;

// provide your workings.
Func<double, double, string> workings = (x, y) =>
    string.Format("The square of {0} is {1}.", x, square(y))
```

Gli oggetti **azione** sono come metodi void quindi hanno solo un tipo di input. Nessun risultato è inserito nello stack di valutazione.

```
// right-angled triangle.
class Triangle
{
    public double a;
    public double b;
    public double h;
}

// Pythagorean theorem.
Action<Triangle> pythagoras = (x) =>
    x.h = squareroot(square(x.a) + square(x.b));

Triangle t = new Triangle { a = 3, b = 4 };
pythagoras(t);
Console.WriteLine(t.h); // 5.
```

Immutabilità

L'immutabilità è comune nella programmazione funzionale e rara nella programmazione orientata agli oggetti.

Creare, ad esempio, un tipo di indirizzo con stato mutabile:

```
public class Address ()
{
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City  { get; set; }
```

```
}
```

Qualsiasi parte di codice potrebbe alterare qualsiasi proprietà nell'oggetto sopra.

Ora crea il tipo di indirizzo immutabile:

```
public class Address ()
{
    public readonly string Line1;
    public readonly string Line2;
    public readonly string City;

    public Address(string line1, string line2, string city)
    {
        Line1 = line1;
        Line2 = line2;
        City  = city;
    }
}
```

Ricordare che disporre di raccolte di sola lettura non rispetta l'immutabilità. Per esempio,

```
public class Classroom
{
    public readonly List<Student> Students;

    public Classroom(List<Student> students)
    {
        Students = students;
    }
}
```

non è immutabile, in quanto l'utente dell'oggetto può modificare la raccolta (aggiungere o rimuovere elementi da essa). Per renderlo immutabile, si deve usare un'interfaccia come IEnumerable, che non espone i metodi da aggiungere, o per renderlo un ReadOnlyCollection.

```
public class Classroom
{
    public readonly ReadOnlyCollection<Student> Students;

    public Classroom(ReadOnlyCollection<Student> students)
    {
        Students = students;
    }
}

List<Students> list = new List<Student>();
// add students
Classroom c = new Classroom(list.AsReadOnly());
```

Con l'oggetto immutabile abbiamo i seguenti vantaggi:

- Sarà in uno stato noto (altro codice non può cambiarlo).
- È thread-safe.
- Il costruttore offre un singolo posto per la convalida.

- Sapere che l'oggetto non può essere modificato rende il codice più facile da capire.

Evita riferimenti null

Gli sviluppatori C # ottengono un sacco di eccezioni di riferimento null da gestire. Gli sviluppatori di F # non lo fanno perché hanno il tipo Option. Un tipo di opzione <> (alcuni preferiscono forse <> come nome) fornisce un tipo di ritorno Some e None. Rende esplicito che un metodo potrebbe essere in procinto di restituire un record nullo.

Ad esempio, non puoi leggere quanto segue e sapere se dovrai gestire un valore nullo.

```
var user = _repository.GetUser(id);
```

Se si conosce l'eventuale null, è possibile introdurre un codice di codice per affrontarlo.

```
var username = user != null ? user.Name : string.Empty;
```

Cosa succede se invece è stata restituita un'opzione <>?

```
Option<User> maybeUser = _repository.GetUser(id);
```

Il codice ora rende esplicito che potremmo avere un record None restituito e il codice boilerplate da verificare per Some o None è richiesto:

```
var username = maybeUser.HasValue ? maybeUser.Value.Name : string.Empty;
```

Il seguente metodo mostra come restituire un'opzione <>

```
public Option<User> GetUser(int id)
{
    var users = new List<User>
    {
        new User { Id = 1, Name = "Joe Bloggs" },
        new User { Id = 2, Name = "John Smith" }
    };

    var user = users.FirstOrDefault(user => user.Id == id);

    return user != null ? new Option<User>(user) : new Option<User>();
}
```

Ecco un'implementazione minima dell'opzione <>.

```
public struct Option<T>
{
    private readonly T _value;

    public T Value
    {
        get
        {
```

```
            if (!HasValue)
                throw new InvalidOperationException();

            return _value;
        }
    }

    public bool HasValue
    {
        get { return _value != null; }
    }

    public Option(T value)
    {
        _value = value;
    }

    public static implicit operator Option<T>(T value)
    {
        return new Option<T>(value);
    }
}
```

Per dimostrare quanto sopra avoidNull.csx può essere eseguito con C # REPL.

Come affermato, questa è un'implementazione minima. Una ricerca di pacchetti "Forse" NuGet mostrerà un buon numero di buone librerie.

Funzioni di ordine superiore

Una funzione di ordine superiore è quella che accetta un'altra funzione come argomento o restituisce una funzione (o entrambe).

Questo è comunemente fatto con lambdas, ad esempio quando si passa un predicato a una clausola LINQ Where:

```
var results = data.Where(p => p.Items == 0);
```

La clausola Where () potrebbe ricevere molti predicati diversi che gli conferiscono una notevole flessibilità.

Passare un metodo in un altro metodo è anche visto quando si implementa il modello di progettazione della strategia. Ad esempio, vari metodi di ordinamento potrebbero essere scelti e passati in un metodo di ordinamento su un oggetto in base ai requisiti in fase di esecuzione.

Collezioni immutabili

Il pacchetto `System.Collections.Immutable` NuGet offre classi di raccolta immutabili.

Creazione e aggiunta di elementi

```
var stack = ImmutableStack.Create<int>();
var stack2 = stack.Push(1); // stack is still empty, stack2 contains 1
var stack3 = stack.Push(2); // stack2 still contains only one, stack3 has 2, 1
```

Creare usando il costruttore

Alcune collezioni immutabili dispongono di una classe interna `Builder` che può essere utilizzata per creare a buon mercato istanze immutabili di grandi dimensioni:

```
var builder = ImmutableList.CreateBuilder<int>(); // returns ImmutableList.Builder
builder.Add(1);
builder.Add(2);
var list = builder.ToImmutable();
```

Creazione da un oggetto I esistente

```
var numbers = Enumerable.Range(1, 5);
var list = ImmutableList.CreateRange<int>(numbers);
```

Elenco di tutti i tipi di raccolta immutabili:

- `System.Collections.Immutable.ImmutableArray<T>`
- `System.Collections.Immutable.ImmutableDictionary<TKey,TValue>`
- `System.Collections.Immutable.ImmutableHashSet<T>`
- `System.Collections.Immutable.ImmutableList<T>`
- `System.Collections.Immutable.ImmutableQueue<T>`
- `System.Collections.Immutable.ImmutableSortedDictionary<TKey,TValue>`
- `System.Collections.Immutable.ImmutableSortedSet<T>`
- `System.Collections.Immutable.ImmutableStack<T>`

Leggi Programmazione funzionale online:
https://riptutorial.com/it/csharp/topic/2564/programmazione-funzionale

Capitolo 125: Programmazione orientata agli oggetti in C

introduzione

Questo argomento prova a dirci come possiamo scrivere programmi basati sull'approccio OOP. Ma non proviamo a insegnare il paradigma della programmazione orientata agli oggetti. Tratteremo i seguenti argomenti: Classi, Proprietà, Ereditarietà, Polimorfismo, Interfacce e così via.

Examples

Classi:

Lo scheletro della classe dichiarante è:

<>: Richiesto

[]:Opzionale

```
[private/public/protected/internal] class <Desired Class Name> [:[Inherited
class][,][[Interface Name 1],[Interface Name 2],...]
{
    //Your code
}
```

Non preoccuparti se non riesci a capire l'intera sintassi, acquisiremo familiarità con tutte le sue parti. Per il primo esempio, considera la seguente classe:

```
class MyClass
{
    int i = 100;
    public void getMyValue()
    {
        Console.WriteLine(this.i);//Will print number 100 in output
    }
}
```

in questa classe creiamo la variabile `i` con tipo `int` e con i Modifier di accesso privati predefiniti e il metodo `getMyValue()` con i modificatori di accesso pubblico.

Leggi Programmazione orientata agli oggetti in C # online:
https://riptutorial.com/it/csharp/topic/9856/programmazione-orientata-agli-oggetti-in-c-sharp

Capitolo 126: Proprietà

Osservazioni

Le proprietà combinano l'archiviazione dei dati di classe dei campi con l'accessibilità dei metodi. A volte può essere difficile decidere se utilizzare una proprietà, una proprietà che fa riferimento a un campo o un metodo che fa riferimento a un campo. Come regola generale:

- Le proprietà dovrebbero essere utilizzate senza un campo interno se ottengono e / o impostano solo valori; senza altra logica che si verifichi. In questi casi, l'aggiunta di un campo interno aggiungerebbe il codice senza alcun beneficio.
- Le proprietà dovrebbero essere utilizzate con i campi interni quando è necessario manipolare o convalidare i dati. Un esempio potrebbe essere la rimozione degli spazi iniziali e finali dalle stringhe o la garanzia che una data non sia nel passato.

Per quanto riguarda Metodi vs Proprietà, dove è possibile recuperare (`get`) e aggiornare (`set`) un valore, una proprietà è la scelta migliore. Inoltre, .Net offre molte funzionalità che fanno uso della struttura di una classe; ad esempio aggiungendo una griglia a un modulo, .Net elenca per default tutte le proprietà della classe su quel modulo; quindi, per utilizzare al meglio tali convenzioni, è consigliabile utilizzare le proprietà quando questo comportamento è in genere auspicabile e i metodi in cui preferiresti che i tipi non vengano aggiunti automaticamente.

Examples

Varie proprietà nel contesto

```
public class Person
{
    //Id property can be read by other classes, but only set by the Person class
    public int Id {get; private set;}
    //Name property can be retrieved or assigned
    public string Name {get; set;}

    private DateTime dob;
    //Date of Birth property is stored in a private variable, but retrieved or assigned
through the public property.
    public DateTime DOB
    {
        get { return this.dob; }
        set { this.dob = value; }
    }
    //Age property can only be retrieved; it's value is derived from the date of birth
    public int Age
    {
        get
        {
            int offset = HasHadBirthdayThisYear() ? 0 : -1;
            return DateTime.UtcNow.Year - this.dob.Year + offset;
        }
```

```
    }

    //this is not a property but a method; though it could be rewritten as a property if
desired.
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}
```

Pubblico Ottieni

I getter sono usati per esporre i valori delle classi.

```
string name;
public string Name
{
    get { return this.name; }
}
```

Set pubblico

I setter sono usati per assegnare valori alle proprietà.

```
string name;
public string Name
{
    set { this.name = value; }
}
```

Accesso alle proprietà

```
class Program
{
    public static void Main(string[] args)
    {
        Person aPerson = new Person("Ann Xena Sample", new DateTime(1984, 10, 22));
        //example of accessing properties (Id, Name & DOB)
```

```
        Console.WriteLine("Id is:  \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());
        //example of setting properties

        aPerson.Name = "   Hans Trimmer  ";
        aPerson.DOB = new DateTime(1961, 11, 11);
        //aPerson.Id = 5; //this won't compile as Id's SET method is private; so only
accessible within the Person class.
        //aPerson.DOB = DateTime.UtcNow.AddYears(1); //this would throw a runtime error as
there's validation to ensure the DOB is in past.

        //see how our changes above take effect; note that the Name has been trimmed
        Console.WriteLine("Id is:  \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());

        Console.WriteLine("Press any key to continue");
        Console.Read();
    }
}

public class Person
{
    private static int nextId = 0;
    private string name;
    private DateTime dob; //dates are held in UTC; i.e. we disregard timezones
    public Person(string name, DateTime dob)
    {
        this.Id = ++Person.nextId;
        this.Name = name;
        this.DOB = dob;
    }
    public int Id
    {
        get;
        private set;
    }
    public string Name
    {
        get { return this.name; }
        set
        {
            if (string.IsNullOrWhiteSpace(value)) throw new InvalidNameException(value);
            this.name = value.Trim();
        }
    }
    public DateTime DOB
    {
        get { return this.dob; }
        set
        {
            if (value < DateTime.UtcNow.AddYears(-200) || value > DateTime.UtcNow) throw new
InvalidDobException(value);
            this.dob = value;
        }
    }
    public int GetAgeInYears()
    {
        DateTime today = DateTime.UtcNow;
        int offset = HasHadBirthdayThisYear() ? 0 : -1;
        return today.Year - this.dob.Year + offset;
    }
```

```
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}

public class InvalidNameException : ApplicationException
{
    const string InvalidNameExceptionMessage = "'{0}' is an invalid name.";
    public InvalidNameException(string value):
base(string.Format(InvalidNameExceptionMessage,value)){}
}
public class InvalidDobException : ApplicationException
{
    const string InvalidDobExceptionMessage = "'{0:yyyy-MM-dd}' is an invalid DOB.  The date
must not be in the future, or over 200 years in the past.";
    public InvalidDobException(DateTime value):
base(string.Format(InvalidDobExceptionMessage,value)){}
}
```

Valori predefiniti per le proprietà

L'impostazione di un valore predefinito può essere eseguita utilizzando Initializers (C # 6)

```
public class Name
{
    public string First { get; set; } = "James";
    public string Last { get; set; } = "Smith";
}
```

Se è letto solo tu puoi restituire valori come questo:

```
  public class Name
  {
      public string First => "James";
      public string Last => "Smith";
  }
```

Proprietà auto-implementate

Le proprietà implementate automaticamente sono state introdotte in C # 3.
Una proprietà auto-implementata è dichiarata con un getter e setter vuoto (accessor):

```
public bool IsValid { get; set; }
```

Quando una proprietà auto-implementata è scritta nel tuo codice, il compilatore crea un campo anonimo privato a cui è possibile accedere solo tramite gli accessors della proprietà.

L'affermazione di proprietà auto-implementata sopra equivale a scrivere questo lungo codice:

```
private bool _isValid;
public bool IsValid
{
    get { return _isValid; }
    set { _isValid = value; }
}
```

Le proprietà implementate automaticamente non possono avere alcuna logica nei loro accessor, ad esempio:

```
public bool IsValid { get; set { PropertyChanged("IsValid"); } } // Invalid code
```

Una proprietà auto-implementata *può* tuttavia avere diversi modificatori di accesso per i suoi accessori:

```
public bool IsValid { get; private set; }
```

Il C # 6 consente alle proprietà auto-implementate di non avere alcun setter (rendendolo immutabile, dal momento che il suo valore può essere impostato solo all'interno del costruttore o hard coded):

```
public bool IsValid { get; }
public bool IsValid { get; } = true;
```

Per ulteriori informazioni sull'inizializzazione delle proprietà autoattive, leggere la documentazione degli inizializzatori della proprietà automatica .

Proprietà di sola lettura

Dichiarazione

Un comune malinteso, in particolare i principianti, è di proprietà di sola `readonly` è quello contrassegnato con una parola chiave `readonly` . Questo non è corretto e in effetti il *seguente è un errore in fase di compilazione* :

```
public readonly string SomeProp { get; set; }
```

Una proprietà è di sola lettura quando ha solo un getter.

```
public string SomeProp { get; }
```

Utilizzo di proprietà di sola lettura per creare classi immutabili

```
public Address
{
    public string ZipCode { get; }
    public string City { get; }
    public string StreetAddress { get; }

    public Address(
        string zipCode,
        string city,
        string streetAddress)
    {
        if (zipCode == null)
            throw new ArgumentNullException(nameof(zipCode));
        if (city == null)
            throw new ArgumentNullException(nameof(city));
        if (streetAddress == null)
            throw new ArgumentNullException(nameof(streetAddress));

        ZipCode = zipCode;
        City = city;
        StreetAddress = streetAddress;
    }
}
```

Leggi Proprietà online: https://riptutorial.com/it/csharp/topic/49/proprieta

Capitolo 127: puntatori

Osservazioni

Puntatori e `unsafe`

A causa della loro natura, i puntatori producono codice non verificabile. Pertanto, l'utilizzo di qualsiasi tipo di puntatore richiede un contesto `unsafe` .

Il tipo `System.IntPtr` è un wrapper sicuro attorno a un `void*` . È inteso come un'alternativa più conveniente a `void*` quando non è richiesto altrimenti un contesto non sicuro per eseguire l'attività a portata di mano.

Comportamento non definito

Come in C e C ++, l'uso scorretto di puntatori può richiamare un comportamento indefinito, con possibili effetti collaterali come il danneggiamento della memoria e l'esecuzione di codice non intenzionale. A causa della natura non verificabile della maggior parte delle operazioni del puntatore, l'uso corretto dei puntatori è interamente una responsabilità del programmatore.

Tipi che supportano i puntatori

A differenza di C e C ++, non tutti i tipi di C # hanno tipi di puntatore corrispondenti. Un tipo `T` può avere un tipo di puntatore corrispondente se si applicano entrambi i seguenti criteri:

- `T` è un tipo di struttura o un tipo di puntatore.
- `T` contiene solo membri che soddisfano entrambi questi criteri in modo ricorsivo.

Examples

Puntatori per l'accesso alla matrice

Questo esempio dimostra come i puntatori possono essere utilizzati per l'accesso di tipo C agli array C #.

```
unsafe
{
    var buffer = new int[1024];
    fixed (int* p = &buffer[0])
    {
        for (var i = 0; i < buffer.Length; i++)
        {
            *(p + i) = i;
```

```
        }
    }
}
```

La parola chiave `unsafe` è necessaria perché l'accesso al puntatore non emetterà alcun controllo sui limiti normalmente emesso quando si accede agli array C # in modo regolare.

La parola chiave `fixed` dice al compilatore C # di emettere istruzioni per bloccare l'oggetto in un modo sicuro. Il blocco è necessario per garantire che il garbage collector non sposterà la matrice in memoria, in quanto ciò invaliderebbe qualsiasi puntatore che punta all'interno dell'array.

Aritmetica del puntatore

L'addizione e la sottrazione nei puntatori funzionano in modo diverso dai numeri interi. Quando un puntatore viene incrementato o decrementato, l'indirizzo a cui punta viene aumentato o diminuito dalla dimensione del tipo di referente.

Ad esempio, il tipo `int` (alias per `System.Int32`) ha una dimensione di 4. Se un `int` può essere memorizzato nell'indirizzo 0, il successivo `int` può essere memorizzato nell'indirizzo 4 e così via. Nel codice:

```
var ptr = (int*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 4
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
```

Analogamente, il tipo `long` (alias per `System.Int64`) ha una dimensione di 8. Se un `long` può essere memorizzato nell'indirizzo 0, il `long` successivo può essere memorizzato nell'indirizzo 8, e così via. Nel codice:

```
var ptr = (long*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 16
```

Il tipo `void` è speciale e anche i puntatori `void` sono speciali e vengono utilizzati come indicatori catch-all quando il tipo non è noto o non ha importanza. A causa della loro natura agnostica, i puntatori di `void` non possono essere incrementati o decrementati:

```
var ptr = (void*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
```

L'asterisco fa parte del tipo

In C e C ++, l'asterisco nella dichiarazione di una variabile puntatore fa *parte dell'espressione* dichiarata. In C #, l'asterisco nella dichiarazione fa *parte del tipo* .

In C, C ++ e C #, il seguente snippet dichiara un puntatore `int` :

```
int* a;
```

In C e C ++, il seguente snippet dichiara un puntatore `int` e una variabile `int` . In C #, dichiara due puntatori `int` :

```
int* a, b;
```

In C e C ++, il seguente snippet dichiara due puntatori `int` . In C #, non è valido:

```
int *a, *b;
```

void *

C # eredita da C e C ++ l'uso di `void*` come puntatore agnostico di tipo e puntatore agnostico.

```
void* ptr;
```

Qualsiasi tipo di puntatore può essere assegnato a `void*` utilizzando una conversione implicita:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
```

Il contrario richiede una conversione esplicita:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
int* p2 = (int*)ptr;
```

Accesso membri usando ->

C # eredita da C e C ++ l'uso del simbolo `->` come mezzo per accedere ai membri di un'istanza tramite un puntatore digitato.

Considera la seguente struttura:

```
struct Vector2
{
    public int X;
    public int Y;
}
```

Questo è un esempio dell'uso di `->` per accedere ai suoi membri:

```
Vector2 v;
v.X = 5;
v.Y = 10;

Vector2* ptr = &v;
int x = ptr->X;
int y = ptr->Y;
string s = ptr->ToString();

Console.WriteLine(x); // prints 5
Console.WriteLine(y); // prints 10
Console.WriteLine(s); // prints Vector2
```

Puntatori generici

I criteri che un tipo deve soddisfare per supportare i puntatori (vedere *Note*) non possono essere espressi in termini di vincoli generici. Pertanto, qualsiasi tentativo di dichiarare un puntatore a un tipo fornito tramite un parametro di tipo generico avrà esito negativo.

```
void P<T>(T obj)
    where T : struct
{
    T* ptr = &obj; // compile-time error
}
```

Leggi puntatori online: https://riptutorial.com/it/csharp/topic/5524/puntatori

Capitolo 128: Puntatori e codice non sicuro

Examples

Introduzione al codice non sicuro

C # consente di utilizzare le variabili del puntatore in una funzione del blocco di codice quando è contrassegnato dal modificatore `unsafe` . Il codice non sicuro o il codice non gestito è un blocco di codice che utilizza una variabile puntatore.

Un puntatore è una variabile il cui valore è l'indirizzo di un'altra variabile, cioè l'indirizzo diretto della posizione di memoria. simile a qualsiasi variabile o costante, è necessario dichiarare un puntatore prima di poterlo utilizzare per memorizzare qualsiasi indirizzo variabile.

La forma generale di una dichiarazione del puntatore è:

```
type *var-name;
```

Di seguito sono riportate le dichiarazioni del puntatore valide:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

Il seguente esempio illustra l'uso di puntatori in C #, usando il modificatore non sicuro:

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      static unsafe void Main(string[] args)
      {
         int var = 20;
         int* p = &var;
         Console.WriteLine("Data is: {0} ",  var);
         Console.WriteLine("Address is: {0}",  (int)p);
         Console.ReadKey();
      }
   }
}
```

Quando il codice sopra riportato viene compilato ed eseguito, produce il seguente risultato:

```
Data is: 20
Address is: 99215364
```

Invece di dichiarare un intero metodo come non sicuro, è anche possibile dichiarare una parte del

codice come non sicura:

```
// safe code
unsafe
{
    // you can use pointers here
}
// safe code
```

Recupero del valore dei dati mediante un puntatore

È possibile recuperare i dati memorizzati nel individuato a cui fa riferimento la variabile puntatore, utilizzando il metodo ToString (). Il seguente esempio dimostra questo:

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      public static void Main()
      {
         unsafe
         {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} " , var);
            Console.WriteLine("Data is: {0} " , p->ToString());
            Console.WriteLine("Address is: {0} " , (int)p);
         }

         Console.ReadKey();
      }
   }
}
```

Quando il codice sopra riportato è stato compilato ed eseguito, produce il seguente risultato:

```
Data is: 20
Data is: 20
Address is: 77128984
```

Passare i puntatori come parametri ai metodi

È possibile passare una variabile puntatore a un metodo come parametro. Il seguente esempio illustra questo:

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe void swap(int* p, int *q)
      {
         int temp = *p;
```

```
         *p = *q;
         *q = temp;
      }

      public unsafe static void Main()
      {
         TestPointer p = new TestPointer();
         int var1 = 10;
         int var2 = 20;
         int* x = &var1;
         int* y = &var2;

         Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
         p.swap(x, y);

         Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
         Console.ReadKey();
      }
   }
}
```

Quando il codice sopra è compilato ed eseguito, produce il seguente risultato:

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

Accedere agli elementi dell'array usando un puntatore

In C #, un nome di matrice e un puntatore a un tipo di dati identico ai dati dell'array, non sono lo stesso tipo di variabile. Ad esempio, `int *p` e `int[] p` , non sono dello stesso tipo. È possibile incrementare la variabile del puntatore `p` perché non è fissata in memoria, ma un indirizzo di array è fisso in memoria e non è possibile incrementarlo.

Pertanto, se è necessario accedere a dati di array utilizzando una variabile puntatore, come facciamo tradizionalmente in C o C ++, è necessario correggere il puntatore utilizzando la parola chiave fixed.

Il seguente esempio dimostra questo:

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe static void Main()
      {
         int[]  list = {10, 100, 200};
         fixed(int *ptr = list)

         /* let us have array address in pointer */
         for ( int i = 0; i < 3; i++)
         {
            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
         }
```

```
            Console.ReadKey();
        }
    }
}
```

Quando il codice sopra riportato è stato compilato ed eseguito, produce il seguente risultato:

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

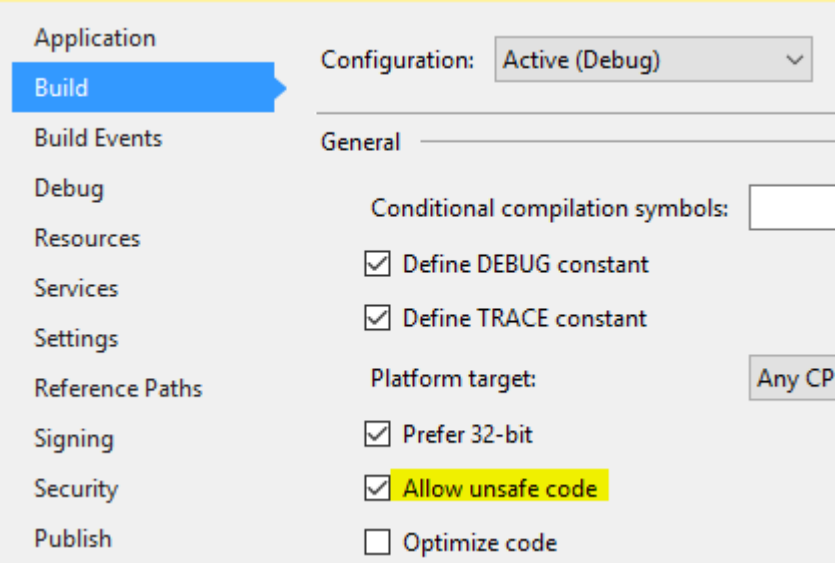
Compilare codice non sicuro

Per la compilazione di codice non sicuro, è necessario specificare l' `/unsafe` riga di comando `/unsafe` con il compilatore della riga di comando.

Ad esempio, per compilare un programma denominato prog1.cs contenente codice non sicuro, dalla riga di comando, dare il comando:

```
csc /unsafe prog1.cs
```

Se si utilizza IDE di Visual Studio, è necessario abilitare l'uso di codice non sicuro nelle proprietà del progetto.



Per fare questo:

- Aprire le proprietà del progetto facendo doppio clic sul nodo delle proprietà in Esplora soluzioni.
- Clicca sulla scheda Costruisci.
- Seleziona l'opzione "Consenti codice non sicuro"

Leggi Puntatori e codice non sicuro online: https://riptutorial.com/it/csharp/topic/5514/puntatori-e-codice-non-sicuro

Capitolo 129: Query LINQ

introduzione

LINQ è un acronimo che sta per **L** anguage **IN** tegrated **Q** uery. È un concetto che integra un linguaggio di query offrendo un modello coerente per lavorare con i dati attraverso vari tipi di fonti e formati di dati; si utilizzano gli stessi schemi di codifica di base per interrogare e trasformare i dati in documenti XML, database SQL, set di dati ADO.NET, raccolte .NET e qualsiasi altro formato per il quale sia disponibile un provider LINQ.

Sintassi

- Sintassi delle query:
    - da <intervallo variabile> in <raccolta>
    - [da <range variable> in <collection>, ...]
    - <filtro, unione, raggruppamento, operatori di aggregazione, ...> <espressione lambda>
    - <selezionare o groupBy operator> <formulare il risultato>
- Sintassi del metodo:
    - Enumerable.Aggregate (func)
    - Enumerable.Aggregate (seed, func)
    - Enumerable.Aggregate (seed, func, resultSelector)
    - Enumerable.All (predicato)
    - Enumerable.Any ()
    - Enumerable.Any (predicato)
    - Enumerable.AsEnumerable ()
    - Enumerable.Average ()
    - Enumerable.Average (selettore)
    - Enumerable.Cast <Risultato> ()
    - Enumerable.Concat (secondo)
    - Enumerable.Contains (valore)
    - Enumerable.Contains (valore, comparatore)
    - Enumerable.Count ()
    - Enumerable.Count (predicato)
    - Enumerable.DefaultIfEmpty ()
    - Enumerable.DefaultIfEmpty (defaultValue)
    - Enumerable.Distinct ()
    - Enumerable.Distinct (di confronto)
    - Enumerable.ElementAt (indice)
    - Enumerable.ElementAtOrDefault (indice)
    - Enumerable.Empty ()
    - Enumerable.Except (secondo)
    - Enumerable.Except (second, comparer)

- Enumerable.First ()
- Enumerable.First (predicato)
- Enumerable.FirstOrDefault ()
- Enumerable.FirstOrDefault (predicato)
- Enumerable.GroupBy (keySelector)
- Enumerable.GroupBy (keySelector, resultSelector)
- Enumerable.GroupBy (keySelector, elementSelector)
- Enumerable.GroupBy (keySelector, comparatore)
- Enumerable.GroupBy (keySelector, resultSelector, comparatore)
- Enumerable.GroupBy (keySelector, elementSelector, resultSelector)
- Enumerable.GroupBy (keySelector, elementSelector, comparatore)
- Enumerable.GroupBy (keySelector, elementSelector, resultSelector, comparatore)
- Enumerable.Intersect (secondo)
- Enumerable.Intersect (second, comparer)
- Enumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector)
- Enumerable.Join (inner, outerKeySelector, innerKeySelector, resultSelector, comparatore)
- Enumerable.Last ()
- Enumerable.Last (predicato)
- Enumerable.LastOrDefault ()
- Enumerable.LastOrDefault (predicato)
- Enumerable.LongCount ()
- Enumerable.LongCount (predicato)
- Enumerable.Max ()
- Enumerable.Max (selettore)
- Enumerable.Min ()
- Enumerable.Min (selettore)
- Enumerable.OfType <TResult> ()
- Enumerable.OrderBy (keySelector)
- Enumerable.OrderBy (keySelector, comparatore)
- Enumerable.OrderByDescending (keySelector)
- Enumerable.OrderByDescending (keySelector, comparatore)
- Enumerable.Range (avvia, conta)
- Enumerable.Repeat (element, count)
- Enumerable.Reverse ()
- Enumerable.Select (selettore)
- Enumerable.SelectMany (selettore)
- Enumerable.SelectMany (collectionSelector, resultSelector)
- Enumerable.SequenceEqual (secondo)
- Enumerable.SequenceEqual (secondo, comparatore)
- Enumerable.Single ()
- Enumerable.Single (predicato)
- Enumerable.SingleOrDefault ()
- Enumerable.SingleOrDefault (predicato)
- Enumerable.Skip (conteggio)
- Enumerable.SkipWhile (predicato)

- Enumerable.Sum ()
- Enumerable.Sum (selettore)
- Enumerable.Take (conteggio)
- Enumerable.TakeWhile (predicato)
- orderedEnumerable.ThenBy (keySelector)
- orderedEnumerable.ThenBy (keySelector, comparatore)
- orderedEnumerable.ThenByDescending (keySelector)
- orderedEnumerable.ThenByDescending (keySelector, comparatore)
- Enumerable.ToArray ()
- Enumerable.ToDictionary (keySelector)
- Enumerable.ToDictionary (keySelector, elementSelector)
- Enumerable.ToDictionary (keySelector, comparatore)
- Enumerable.ToDictionary (keySelector, elementSelector, comparatore)
- Enumerable.ToList ()
- Enumerable.ToLookup (keySelector)
- Enumerable.ToLookup (keySelector, elementSelector)
- Enumerable.ToLookup (keySelector, comparatore)
- Enumerable.ToLookup (keySelector, elementSelector, comparatore)
- Enumerable.Union (secondo)
- Enumerable.Union (second, comparer)
- Enumerable.Where (predicato)
- Enumerable.Zip (second, resultSelector)

Osservazioni

Per utilizzare le query LINQ è necessario importare `System.Linq` .

La sintassi del metodo è più potente e flessibile, ma la sintassi delle query potrebbe essere più semplice e più familiare. Tutte le query scritte nella sintassi Query sono tradotte nella sintassi funzionale dal compilatore, quindi le prestazioni sono le stesse.

Gli oggetti di query non vengono valutati fino a quando non vengono utilizzati, in modo che possano essere modificati o aggiunti senza penalizzare le prestazioni.

Examples

Dove

Restituisce un sottoinsieme di elementi che il predicato specificato è vero per loro.

```
List<string> trees = new List<string>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Sintassi del metodo

```
// Select all trees with name of length 3
```

```
var shortTrees = trees.Where(tree => tree.Length == 3); // Oak, Elm
```

Sintassi delle query

```
var shortTrees = from tree in trees
                 where tree.Length == 3
                 select tree; // Oak, Elm
```

Seleziona - Trasforma elementi

Select consente di applicare una trasformazione a ogni elemento in qualsiasi struttura dati che implementa IEnumerable.

Ottenere il primo carattere di ogni stringa nel seguente elenco:

```
List<String> trees = new List<String>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Usando la sintassi normale (lambda)

```
//The below select stament transforms each element in tree into its first character.
IEnumerable<String> initials = trees.Select(tree => tree.Substring(0, 1));
foreach (String initial in initials) {
    System.Console.WriteLine(initial);
}
```

Produzione:

O
B
B
E
H
M

Live Demo su .NET Fiddle

Utilizzando la sintassi Query LINQ

```
initials = from tree in trees
           select tree.Substring(0, 1);
```

Metodi di concatenamento

Molte funzioni LINQ funzionano entrambe su un oggetto `IEnumerable<TSource>` e restituiscono anche un `IEnumerable<TResult>` . I parametri di tipo `TSource` e `TResult` possono o meno riferirsi allo stesso tipo, a seconda del metodo in questione e delle eventuali funzioni passate ad esso.

Alcuni esempi di questo sono

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

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate
)

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

Mentre alcuni metodi di concatenamento possono richiedere l' esecuzione di un intero set prima di andare avanti, LINQ sfrutta l' esecuzione differita usando **MSDN** return return che crea un Enumerable e un Enumerator dietro le quinte. Il processo di concatenamento in LINQ consiste essenzialmente nella costruzione di un enumerabile (iteratore) per il set originale - che viene rinviato - fino a quando non si materializza enumerando l'enumerabile .

Ciò consente a queste funzioni di essere fluentemente collegate a **wiki** , in cui una funzione può agire direttamente sul risultato di un'altra. Questo stile di codice può essere utilizzato per eseguire molte operazioni basate su sequenze in una singola istruzione.

Ad esempio, è possibile combinare `Select` , `Where` e `OrderBy` per trasformare, filtrare e ordinare una sequenza in una singola istruzione.

```
var someNumbers = { 4, 3, 2, 1 };

var processed = someNumbers
        .Select(n => n * 2)    // Multiply each number by 2
        .Where(n => n != 6)    // Keep all the results, except for 6
        .OrderBy(n => n);      // Sort in ascending order
```

Produzione:

2
4
8

Live Demo su .NET Fiddle

Qualsiasi funzione che estenda e restituisca il tipo `IEnumerable<T>` generico può essere utilizzata come clausole concatenate in una singola istruzione. Questo stile di programmazione fluida è potente e dovrebbe essere preso in considerazione quando si creano i propri metodi di estensione .

Intervallo e ripetizione

I metodi statici `Range` e `Repeat` su `Enumerable` possono essere utilizzati per generare sequenze semplici.

Gamma

`Enumerable.Range()` genera una sequenza di numeri interi dati un valore iniziale e un conteggio.

```
// Generate a collection containing the numbers 1-100 ([1, 2, 3, ..., 98, 99, 100])
var range = Enumerable.Range(1,100);
```

Live Demo su .NET Fiddle

Ripetere

`Enumerable.Repeat()` genera una sequenza di elementi ripetuti in base a un elemento e al numero di ripetizioni richieste.

```
// Generate a collection containing "a", three times (["a","a","a"])
var repeatedValues = Enumerable.Repeat("a", 3);
```

Live Demo su .NET Fiddle

Salta e prendi

Il metodo Skip restituisce una collezione escludendo un numero di elementi dall'inizio della raccolta di origine. Il numero di articoli esclusi è il numero dato come argomento. Se nella raccolta sono presenti meno elementi di quelli specificati nell'argomento, viene restituita una raccolta vuota.

Il metodo Take restituisce una raccolta contenente un numero di elementi dall'inizio della raccolta di origine. Il numero di elementi inclusi è il numero indicato come argomento. Se nella raccolta sono presenti meno elementi di quelli specificati nell'argomento, la raccolta restituita conterrà gli stessi elementi della raccolta di origine.

```
var values = new [] { 5, 4, 3, 2, 1 };

var skipTwo        = values.Skip(2);         // { 3, 2, 1 }
var takeThree      = values.Take(3);         // { 5, 4, 3 }
var skipOneTakeTwo = values.Skip(1).Take(2); // { 4, 3 }
var takeZero       = values.Take(0);         // An IEnumerable<int> with 0 items
```

Live Demo su .NET Fiddle

Skip and Take sono comunemente usati insieme per impaginare i risultati, ad esempio:

```
IEnumerable<T> GetPage<T>(IEnumerable<T> collection, int pageNumber, int resultsPerPage) {
    int startIndex = (pageNumber - 1) * resultsPerPage;
    return collection.Skip(startIndex).Take(resultsPerPage);
}
```

Avviso: LINQ to Entities supporta solo Salta sulle query ordinate . Se si tenta di

utilizzare Skip senza ordinare, verrà visualizzato **NotSupportedException** con il messaggio "Il metodo 'Skip' è supportato solo per l'input ordinato in LINQ alle entità. Il metodo 'OrderBy' deve essere chiamato prima del metodo 'Salta'."

Innanzitutto, FirstOrDefault, Last, LastOrDefault, Single e SingleOrDefault

Tutti e sei i metodi restituiscono un singolo valore del tipo di sequenza e possono essere chiamati con o senza un predicato.

A seconda del numero di elementi che corrispondono al `predicate` o, se non viene fornito alcun `predicate` , il numero di elementi nella sequenza sorgente, si comportano come segue:

Primo()

- Restituisce il primo elemento di una sequenza o il primo elemento che corrisponde al `predicate` fornito.
- Se la sequenza non contiene elementi, viene generata una `InvalidOperationException` con il messaggio: "Sequenza non contiene elementi".
- Se la sequenza non contiene elementi corrispondenti al `predicate` fornito, viene generata una `InvalidOperationException` con il messaggio "Sequenza non contiene alcun elemento corrispondente".

Esempio

```
// Returns "a":
new[] { "a" }.First();

// Returns "a":
new[] { "a", "b" }.First();

// Returns "b":
new[] { "a", "b" }.First(x => x.Equals("b"));

// Returns "ba":
new[] { "ba", "be" }.First(x => x.Contains("b"));

// Throws InvalidOperationException:
new[] { "ca", "ce" }.First(x => x.Contains("b"));

// Throws InvalidOperationException:
new string[0].First();
```

Live Demo su .NET Fiddle

FirstOrDefault ()

- Restituisce il primo elemento di una sequenza o il primo elemento che corrisponde al `predicate` fornito.
- Se la sequenza non contiene elementi o nessun elemento che corrisponde al `predicate`

fornito, restituisce il valore predefinito del tipo di sequenza usando il `default(T)` .

Esempio

```
// Returns "a":
new[] { "a" }.FirstOrDefault();

// Returns "a":
new[] { "a", "b" }.FirstOrDefault();

// Returns "b":
new[] { "a", "b" }.FirstOrDefault(x => x.Equals("b"));

// Returns "ba":
new[] { "ba", "be" }.FirstOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.FirstOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].FirstOrDefault();
```

Live Demo su .NET Fiddle

Scorso()

- Restituisce l'ultimo elemento di una sequenza o l'ultimo elemento che corrisponde al `predicate` fornito.
- Se la sequenza non contiene elementi, viene lanciata una `InvalidOperationException` con il messaggio "Sequenza non contiene elementi".
- Se la sequenza non contiene elementi corrispondenti al `predicate` fornito, viene generata una `InvalidOperationException` con il messaggio "Sequenza non contiene alcun elemento corrispondente".

Esempio

```
// Returns "a":
new[] { "a" }.Last();

// Returns "b":
new[] { "a", "b" }.Last();

// Returns "a":
new[] { "a", "b" }.Last(x => x.Equals("a"));

// Returns "be":
new[] { "ba", "be" }.Last(x => x.Contains("b"));

// Throws InvalidOperationException:
new[] { "ca", "ce" }.Last(x => x.Contains("b"));

// Throws InvalidOperationException:
new string[0].Last();
```

LastOrDefault ()

- Restituisce l'ultimo elemento di una sequenza o l'ultimo elemento che corrisponde al `predicate` fornito.
- Se la sequenza non contiene elementi o nessun elemento che corrisponde al `predicate` fornito, restituisce il valore predefinito del tipo di sequenza usando il `default(T)` .

Esempio

```
// Returns "a":
new[] { "a" }.LastOrDefault();

// Returns "b":
new[] { "a", "b" }.LastOrDefault();

// Returns "a":
new[] { "a", "b" }.LastOrDefault(x => x.Equals("a"));

 // Returns "be":
new[] { "ba", "be" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].LastOrDefault();
```

Singolo ()

- Se la sequenza contiene esattamente un elemento, o esattamente un elemento che corrisponde al `predicate` fornito, tale elemento viene restituito.
- Se la sequenza non contiene elementi o nessun elemento corrispondente al `predicate` fornito, viene generata una `InvalidOperationException` con il messaggio "Sequenza non contiene elementi".
- Se la sequenza contiene più di un elemento o più di un elemento che corrisponde al `predicate` fornito, viene generata una `InvalidOperationException` con il messaggio "Sequenza contiene più di un elemento".
- **Nota:** per valutare se la sequenza contiene esattamente un elemento, al massimo due elementi devono essere enumerati.

Esempio

```
// Returns "a":
new[] { "a" }.Single();

// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "b" }.Single();

// Returns "b":
new[] { "a", "b" }.Single(x => x.Equals("b"));
```

```
// Throws InvalidOperationException:
new[] { "a", "b" }.Single(x => x.Equals("c"));

// Throws InvalidOperationException:
new string[0].Single();

// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "a" }.Single();
```

SingleOrDefault ()

- Se la sequenza contiene esattamente un elemento, o esattamente un elemento che corrisponde al `predicate` fornito, tale elemento viene restituito.
- Se la sequenza non contiene elementi o nessun elemento corrispondente al `predicate` fornito, viene restituito il `default(T)` .
- Se la sequenza contiene più di un elemento o più di un elemento che corrisponde al `predicate` fornito, viene generata una `InvalidOperationException` con il messaggio "Sequenza contiene più di un elemento".
- Se la sequenza non contiene elementi corrispondenti al `predicate` fornito, restituisce il valore predefinito del tipo di sequenza usando il `default(T)` .
- **Nota:** per valutare se la sequenza contiene esattamente un elemento, al massimo due elementi devono essere enumerati.

Esempio

```
// Returns "a":
new[] { "a" }.SingleOrDefault();

// returns "a"
new[] { "a", "b" }.SingleOrDefault(x => x == "a");

// Returns null:
new[] { "a", "b" }.SingleOrDefault(x => x == "c");

// Throws InvalidOperationException:
new[] { "a", "a" }.SingleOrDefault(x => x == "a");

// Throws InvalidOperationException:
new[] { "a", "b" }.SingleOrDefault();

// Returns null:
new string[0].SingleOrDefault();
```

raccomandazioni

- Sebbene sia possibile utilizzare `FirstOrDefault` , `LastOrDefault` o `SingleOrDefault` per verificare se una sequenza contiene elementi, `Any` o `Count` sono più affidabili. Questo perché un valore di ritorno di `default(T)` da uno di questi tre metodi non dimostra che la sequenza sia vuota, poiché il valore del primo / ultimo / singolo elemento della sequenza potrebbe

essere ugualmente `default(T)`

- Decidi quali metodi si adattano di più al tuo codice. Ad esempio, usa `Single` solo se devi assicurarti che ci sia un singolo oggetto nella collezione che corrisponde al tuo predicato, altrimenti usa `First` ; come `Single` lancia un'eccezione se la sequenza ha più di un elemento corrispondente. Questo ovviamente vale anche per le controparti "* OrDefault".
- Riguardo all'efficienza: sebbene sia spesso opportuno assicurarsi che vi sia un solo elemento (`Single`) o, o solo uno o zero (`SingleOrDefault`), restituiti da una query, entrambi questi metodi richiedono più, e spesso l'intero, della raccolta da esaminare per assicurarsi che non ci sia una seconda corrispondenza alla domanda. Questo è diverso dal comportamento di, ad esempio, il `First` metodo, che può essere soddisfatto dopo aver trovato la prima corrispondenza.

tranne

Il metodo Except restituisce l'insieme di elementi contenuti nella prima raccolta ma non contenuti nella seconda. Il valore predefinito di `IEqualityComparer` viene utilizzato per confrontare gli elementi all'interno dei due set. C'è un sovraccarico che accetta un `IEqualityComparer` come argomento.

Esempio:

```
int[] first = { 1, 2, 3, 4 };
int[] second = { 0, 2, 3, 5 };

IEnumerable<int> inFirstButNotInSecond = first.Except(second);
// inFirstButNotInSecond = { 1, 4 }
```

Produzione:

1
4

Live Demo su .NET Fiddle

In questo caso. `.Except(second)` esclude gli elementi contenuti nell'array `second` , ovvero 2 e 3 (0 e 5 non sono contenuti nel `first` array e vengono saltati).

Nota che `Except` implica `Distinct` (cioè rimuove gli elementi ripetuti). Per esempio:

```
int[] third = { 1, 1, 1, 2, 3, 4 };

IEnumerable<int> inThirdButNotInSecond = third.Except(second);
// inThirdButNotInSecond = { 1, 4 }
```

Produzione:

1
4

Live Demo su .NET Fiddle

In questo caso, gli elementi 1 e 4 vengono restituiti una sola volta.

Implementando `IEquatable` o fornendo la funzione a `IEqualityComparer` consentirà di utilizzare un metodo diverso per confrontare gli elementi. Si noti che il metodo `GetHashCode` deve essere sovrascritto in modo che restituisca un codice hash identico per l' `object` identico in base all'implementazione `IEquatable` .

Esempio con IEquatable:

```
class Holiday : IEquatable<Holiday>
{
    public string Name { get; set; }

    public bool Equals(Holiday other)
    {
        return Name == other.Name;
    }

    // GetHashCode must return true whenever Equals returns true.
    public override int GetHashCode()
    {
        //Get hash code for the Name field if it is not null.
        return Name?.GetHashCode() ?? 0;
    }
}

public class Program
{
    public static void Main()
    {
        List<Holiday> holidayDifference = new List<Holiday>();

        List<Holiday> remoteHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Hanukkah" },
            new Holiday { Name = "Ramadan" }
        };

        List<Holiday> localHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Ramadan" }
        };

        holidayDifference = remoteHolidays
            .Except(localHolidays)
            .ToList();

        holidayDifference.ForEach(x => Console.WriteLine(x.Name));
    }
}
```

Produzione:

Hanukkah

Live Demo su .NET Fiddle

SelectMany: appiattimento di una sequenza di sequenze

```
var sequenceOfSequences = new [] { new [] { 1, 2, 3 }, new [] { 4, 5 }, new [] { 6 } };
var sequence = sequenceOfSequences.SelectMany(x => x);
// returns { 1, 2, 3, 4, 5, 6 }
```

Utilizzare `SelectMany()` se si dispone, o si sta creando una sequenza di sequenze, ma si desidera il risultato come una lunga sequenza.

In LINQ Query Sintassi:

```
var sequence = from subSequence in sequenceOfSequences
               from item in subSequence
               select item;
```

Se si dispone di una raccolta di raccolte e si desidera poter lavorare contemporaneamente ai dati dalla raccolta padre e figlio, è anche possibile con `SelectMany` .

Definiamo classi semplici

```
public class BlogPost
{
    public int Id { get; set; }
    public string Content { get; set; }
    public List<Comment> Comments { get; set; }
}

public class Comment
{
    public int Id { get; set; }
    public string Content { get; set; }
}
```

Supponiamo di avere la seguente collezione.

```
List<BlogPost> posts = new List<BlogPost>()
{
    new BlogPost()
    {
        Id = 1,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 1,
                Content = "It's really great!",
            },
            new Comment()
            {
                Id = 2,
```

```
                Content = "Cool post!"
            }
        }
    },
    new BlogPost()
    {
        Id = 2,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 3,
                Content = "I don't think you're right",
            },
            new Comment()
            {
                Id = 4,
                Content = "This post is a complete nonsense"
            }
        }
    }
};
```

Ora vogliamo selezionare i commenti `Content` con `Id` di `BlogPost` associata a questo commento. Per fare ciò, possiamo usare il sovraccarico `SelectMany` appropriato.

```
var commentsWithIds = posts.SelectMany(p => p.Comments, (post, comment) => new { PostId =
post.Id, CommentContent = comment.Content });
```

I nostri `commentsWithIds` assomiglia a questo

```
{
    PostId = 1,
    CommentContent = "It's really great!"
},
{
    PostId = 1,
    CommentContent = "Cool post!"
},
{
    PostId = 2,
    CommentContent = "I don't think you're right"
},
{
    PostId = 2,
    CommentContent = "This post is a complete nonsense"
}
```

SelectMany

Il metodo SelectMany linq 'appiattisce' un oggetto `IEnumerable<IEnumerable<T>>` in `IEnumerable<T>` . Tutti gli elementi T all'interno delle istanze `IEnumerable` contenute nella sorgente `IEnumerable` verranno combinati in un singolo oggetto `IEnumerable` .

```
var words = new [] { "a,b,c", "d,e", "f" };
```

```
var splitAndCombine = words.SelectMany(x => x.Split(','));
// returns { "a", "b", "c", "d", "e", "f" }
```

Se si utilizza una funzione di selezione che trasforma gli elementi di input in sequenze, il risultato saranno gli elementi di tali sequenze restituite una alla volta.

Si noti che, diversamente da `Select()` , il numero di elementi nell'output non deve necessariamente essere lo stesso dell'input.

Più esempio del mondo reale

```
class School
{
    public Student[] Students { get; set; }
}

class Student
{
    public string Name { get; set; }
}

var schools = new [] {
    new School(){ Students = new [] { new Student { Name="Bob"}, new Student { Name="Jack"}
}},
    new School(){ Students = new [] { new Student { Name="Jim"}, new Student { Name="John"} }}
};

var allStudents = schools.SelectMany(s=> s.Students);

foreach(var student in allStudents)
{
    Console.WriteLine(student.Name);
}
```

Produzione:

peso
Jack
Jim
John

Live Demo su .NET Fiddle

Tutti

`All` è usato per controllare, se tutti gli elementi di una collezione corrispondono a una condizione o meno.
vedi anche:. Any

1. Parametro vuoto

Tutto : non è consentito l'uso con parametro vuoto.

2. Espressione lambda come parametro

Tutto : restituisce `true` se tutti gli elementi della raccolta soddisfano l'espressione lambda e `false` altrimenti:

```
var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.All(i => i < 10); // true
bool result = numbers.All(i => i >= 3); // false
```

3. Raccolta vuota

Tutto : restituisce `true` se la raccolta è vuota e viene fornita un'espressione lambda:

```
var numbers = new List<int>();
bool result = numbers.All(i => i >= 0); // true
```

Nota: `All` interromperanno l'iterazione della raccolta non appena trova un elemento che **non** corrisponde alla condizione. Ciò significa che la raccolta non sarà necessariamente completamente elencata; sarà elencato solo abbastanza lontano da trovare il primo elemento che **non corrisponde** alla condizione.

Query raccolta per tipo / cast elementi da digitare

```
interface IFoo { }
class Foo : IFoo { }
class Bar : IFoo { }
```

```
var item0 = new Foo();
var item1 = new Foo();
var item2 = new Bar();
var item3 = new Bar();
var collection = new IFoo[] { item0, item1, item2, item3 };
```

Utilizzando `OfType`

```
var foos = collection.OfType<Foo>(); // result: IEnumerable<Foo> with item0 and item1
var bars = collection.OfType<Bar>(); // result: IEnumerable<Bar> item item2 and item3
var foosAndBars = collection.OfType<IFoo>(); // result: IEnumerable<IFoo> with all four items
```

Usando `Where`

```
var foos = collection.Where(item => item is Foo); // result: IEnumerable<IFoo> with item0 and
item1
var bars = collection.Where(item => item is Bar); // result: IEnumerable<IFoo> with item2 and
item3
```

Utilizzando `Cast`

```
var bars = collection.Cast<Bar>();                // throws InvalidCastException on the 1st
item
var foos = collection.Cast<Foo>();                // throws InvalidCastException on the 3rd
item
var foosAndBars = collection.Cast<IFoo>();        // OK
```

Unione

Unisce due raccolte per creare una raccolta distinta utilizzando il comparatore di uguaglianza predefinito

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 2, 3, 4, 5 };

var allElement = numbers1.Union(numbers2);   // AllElement now contains 1,2,3,4,5
```

Live Demo su .NET Fiddle

SI UNISCE

I join vengono utilizzati per combinare diversi elenchi o tabelle contenenti i dati tramite una chiave comune.

Come in SQL, i seguenti tipi di join sono supportati in LINQ:
Giunzioni esterne interne, a sinistra, a destra, a croce e **complete** .

I seguenti due elenchi sono utilizzati negli esempi seguenti:

```
var first = new List<string>(){ "a","b","c"}; // Left data
var second = new List<string>(){ "a", "c", "d"}; // Right data
```

(Interno) Unisciti

```
var result = from f in first
             join s in second on f equals s
             select new { f, s };

var result = first.Join(second,
                        f => f,
                        s => s,
                        (f, s) => new { f, s });

// Result: {"a","a"}
//         {"c","c"}
```

Giuntura esterna sinistra

```
var leftOuterJoin = from f in first
                    join s in second on f equals s into temp
                    from t in temp.DefaultIfEmpty()
```

```
                    select new { First = f, Second = t};

// Or can also do:
var leftOuterJoin = from f in first
                    from s in second.Where(x => x == f).DefaultIfEmpty()
                    select new { First = f, Second = s};

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}


// Left outer join method syntax
var leftOuterJoinFluentSyntax = first.GroupJoin(second,
                                      f => f,
                                      s => s,
                                      (f, s) => new { First = f, Second = s })
                                    .SelectMany(temp => temp.Second.DefaultIfEmpty(),
                                      (f, s) => new { First = f.First, Second = s });
```

Giusto outer join

```
var rightOuterJoin = from s in second
                     join f in first on s equals f into temp
                     from t in temp.DefaultIfEmpty()
                     select new {First=t,Second=s};

// Result: {"a","a"}
//         {"c","c"}
//         {null,"d"}
```

Cross Join

```
var CrossJoin = from f in first
                from s in second
                select new { f, s };

// Result: {"a","a"}
//         {"a","c"}
//         {"a","d"}
//         {"b","a"}
//         {"b","c"}
//         {"b","d"}
//         {"c","a"}
//         {"c","c"}
//         {"c","d"}
```

Full Outer Join

```
var fullOuterjoin = leftOuterJoin.Union(rightOuterJoin);

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}
```

```
//         {null,"d"}
```

Esempio pratico

Gli esempi sopra hanno una struttura dati semplice in modo che tu possa concentrarti sulla comprensione dei diversi LINQ che si uniscono tecnicamente, ma nel mondo reale avresti tabelle con colonne a cui devi unirti.

Nell'esempio seguente, vi è una sola `Region` classe utilizzata, in realtà si unirebbero due o più tabelle diverse che contengono la stessa chiave (in questo esempio, il `first` e il `second` vengono uniti tramite l' `ID` chiave comune).

Esempio: considerare la seguente struttura di dati:

```
public class Region
{
    public Int32 ID;
    public string RegionDescription;

    public Region(Int32 pRegionID, string pRegionDescription=null)
    {
        ID = pRegionID; RegionDescription = pRegionDescription;
    }
}
```

Ora prepara i dati (cioè compila con i dati):

```
// Left data
var first = new List<Region>()
                { new Region(1), new Region(3), new Region(4) };
// Right data
var second = new List<Region>()
                {
                   new Region(1, "Eastern"),  new Region(2, "Western"),
                   new Region(3, "Northern"), new Region(4, "Southern")
                };
```

Puoi vedere che in questo esempio `first` non contiene alcuna descrizione di regione, quindi vuoi unirti a loro dal `second` . Quindi l'unione interna sarebbe simile a:

```
// do the inner join
var result = from f in first
             join s in second on f.ID equals s.ID
             select new { f.ID, s.RegionDescription };


 // Result: {1,"Eastern"}
 //         {3, Northern}
 //         {4,"Southern"}
```

Questo risultato ha creato oggetti anonimi al volo, il che va bene, ma abbiamo già creato una classe appropriata, quindi possiamo specificarla: invece di `select new { f.ID, s.RegionDescription };`

possiamo dire `select new Region(f.ID, s.RegionDescription);` , che restituirà gli stessi dati ma creerà oggetti di tipo `Region` , che manterranno la compatibilità con gli altri oggetti.

Demo live su .NET fiddle

distinto

Restituisce valori univoci da un oggetto `IEnumerable` . L'univocità viene determinata utilizzando il comparatore di uguaglianza predefinito.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

var distinct = array.Distinct();
// distinct = { 1, 2, 3, 4, 5 }
```

Per confrontare un tipo di dati personalizzato, è necessario implementare l' `IEquatable<T>` e fornire i metodi `GetHashCode` ed `Equals` per il tipo. Oppure il comparatore di uguaglianza può essere sovrascritto:

```
class SSNEqualityComparer : IEqualityComparer<Person> {
    public bool Equals(Person a, Person b) => return a.SSN == b.SSN;
    public int GetHashCode(Person p) => p.SSN;
}

List<Person> people;

distinct = people.Distinct(SSNEqualityComparer);
```

Raggruppa uno o più campi

Supponiamo che abbiamo qualche modello di film:

```
public class Film {
    public string Title { get; set; }
    public string Category { get; set; }
    public int Year { get; set; }
}
```

Raggruppa per categoria proprietà:

```
foreach (var grp in films.GroupBy(f => f.Category)) {
    var groupCategory = grp.Key;
    var numberOfFilmsInCategory = grp.Count();
}
```

Raggruppa per categoria e anno:

```
foreach (var grp in films.GroupBy(f => new { Category = f.Category, Year = f.Year })) {
    var groupCategory = grp.Key.Category;
    var groupYear = grp.Key.Year;
    var numberOfFilmsInCategory = grp.Count();
}
```

Utilizzo di Range con vari metodi Linq

È possibile utilizzare la classe Enumerable insieme alle query Linq per convertire i loop in Linq one liners.

Seleziona Esempio

Opposto a questo:

```
var asciiCharacters = new List<char>();
for (var x = 0; x < 256; x++)
{
    asciiCharacters.Add((char)x);
}
```

Puoi farlo:

```
var asciiCharacters = Enumerable.Range(0, 256).Select(a => (char) a);
```

Dove esempio

In questo esempio, verranno generati 100 numeri e anche quelli verranno estratti

```
var evenNumbers = Enumerable.Range(1, 100).Where(a => a % 2 == 0);
```

Ordinamento delle query - OrderBy () ThenBy () OrderByDescending () ThenByDescending ()

```
string[] names= { "mark", "steve", "adam" };
```

Ascendente:

Sintassi delle query

```
var sortedNames =
    from name in names
    orderby name
    select name;
```

Sintassi del metodo

```
var sortedNames = names.OrderBy(name => name);
```

sortedNames contiene i nomi nell'ordine seguente: "adam", "mark", "steve"

Discendente:

Sintassi delle query

```
var sortedNames =
    from name in names
    orderby name descending
    select name;
```

Sintassi del metodo

```
var sortedNames = names.OrderByDescending(name => name);
```

sortedNames contiene i nomi nell'ordine seguente: "steve", "mark", "adam"

Ordina per più campi

```
Person[] people =
{
    new Person { FirstName = "Steve", LastName = "Collins", Age = 30},
    new Person { FirstName = "Phil" , LastName = "Collins", Age = 28},
    new Person { FirstName = "Adam" , LastName = "Ackerman", Age = 29},
    new Person { FirstName = "Adam" , LastName = "Ackerman", Age = 15}
};
```

Sintassi delle query

```
var sortedPeople = from person in people
                   orderby person.LastName, person.FirstName, person.Age descending
                   select person;
```

Sintassi del metodo

```
 sortedPeople = people.OrderBy(person => person.LastName)
                      .ThenBy(person => person.FirstName)
                      .ThenByDescending(person => person.Age);
```

Risultato

```
1. Adam Ackerman 29
2. Adam Ackerman 15
3. Phil Collins  28
4. Steve Collins 30
```

Nozioni di base

LINQ è in gran parte utile per l'interrogazione di collezioni (o matrici).

Ad esempio, dati i seguenti dati di esempio:

```
var classroom = new Classroom
{
    new Student { Name = "Alice", Grade = 97, HasSnack = true  },
    new Student { Name = "Bob",   Grade = 82, HasSnack = false },
    new Student { Name = "Jimmy", Grade = 71, HasSnack = true  },
    new Student { Name = "Greg",  Grade = 90, HasSnack = false },
```

```
    new Student { Name = "Joe",   Grade = 59, HasSnack = false }
}
```

Possiamo "interrogare" su questi dati usando la sintassi LINQ. Ad esempio, per recuperare tutti gli studenti che hanno uno spuntino oggi:

```
var studentsWithSnacks = from s in classroom.Students
                         where s.HasSnack
                         select s;
```

Oppure, per recuperare studenti con un punteggio pari o superiore a 90, e restituire solo i loro nomi, non l'oggetto `Student` completo:

```
var topStudentNames = from s in classroom.Students
                      where s.Grade >= 90
                      select s.Name;
```

La funzione LINQ è composta da due sintassi che svolgono le stesse funzioni, hanno prestazioni quasi identiche, ma sono scritte in modo molto diverso. La sintassi dell'esempio precedente è chiamata **sintassi della query** . L'esempio seguente, tuttavia, illustra la **sintassi del metodo** . Gli stessi dati verranno restituiti come nell'esempio sopra, ma il modo in cui la query è scritta è diverso.

```
var topStudentNames = classroom.Students
                              .Where(s => s.Grade >= 90)
                              .Select(s => s.Name);
```

Raggruppa per

GroupBy è un modo semplice per ordinare una raccolta di oggetti `IEnumerable<T>` in gruppi distinti.

Semplice esempio

In questo primo esempio, ci ritroviamo con due gruppi, elementi pari e dispari.

```
List<int> iList = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var grouped = iList.GroupBy(x => x % 2 == 0);

//Groups iList into odd [13579] and even[2468] items

foreach(var group in grouped)
{
    foreach (int item in group)
    {
        Console.Write(item); // 135792468  (first odd then even)
    }
}
```

Esempio più complesso

Prendiamo come esempio un elenco di persone per età. Per prima cosa creeremo un oggetto Persona che ha due proprietà, Nome ed Età.

```
public class Person
{
    public int Age {get; set;}
    public string Name {get; set;}
}
```

Quindi creiamo la nostra lista di esempi di persone con vari nomi ed età.

```
List<Person> people = new List<Person>();
people.Add(new Person{Age = 20, Name = "Mouse"});
people.Add(new Person{Age = 30, Name = "Neo"});
people.Add(new Person{Age = 40, Name = "Morpheus"});
people.Add(new Person{Age = 30, Name = "Trinity"});
people.Add(new Person{Age = 40, Name = "Dozer"});
people.Add(new Person{Age = 40, Name = "Smith"});
```

Quindi creiamo una query LINQ per raggruppare il nostro elenco di persone per età.

```
var query = people.GroupBy(x => x.Age);
```

In questo modo, possiamo vedere l'età per ogni gruppo e avere un elenco di ogni persona nel gruppo.

```
foreach(var result in query)
{
    Console.WriteLine(result.Key);

    foreach(var person in result)
        Console.WriteLine(person.Name);
}
```

Ciò risulta nell'output seguente:

```
20
Mouse
30
Neo
Trinity
40
Morpheus
Dozer
Smith
```

Puoi giocare con la demo live su .NET Fiddle

Qualunque

`Any` viene usato per verificare se **qualche** elemento di una collezione corrisponde a una condizione o meno.

vedi anche: .Tutte , ne e FirstOrDefault: best practice

1. Parametro vuoto

Qualsiasi : restituisce `true` se la raccolta ha elementi e `false` se la raccolta è vuota:

```
var numbers = new List<int>();
bool result = numbers.Any(); // false

var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.Any(); //true
```

2. Espressione lambda come parametro

Qualsiasi : restituisce `true` se la raccolta ha uno o più elementi che soddisfano la condizione nell'espressione lambda:

```
var arrayOfStrings = new string[] { "a", "b", "c" };
arrayOfStrings.Any(item => item == "a");    // true
arrayOfStrings.Any(item => item == "d");    // false
```

3. Raccolta vuota

Qualsiasi : restituisce `false` se la raccolta è vuota e viene fornita un'espressione lambda:

```
var numbers = new List<int>();
bool result = numbers.Any(i => i >= 0); // false
```

Nota: `Any` interromperà l'iterazione della raccolta non appena trova un elemento corrispondente alla condizione. Ciò significa che la raccolta non sarà necessariamente completamente elencata; verrà elencato solo quanto basta per trovare il primo elemento corrispondente alla condizione.

Live Demo su .NET Fiddle

ToDictionary

Il metodo LINQ `ToDictionary()` può essere utilizzato per generare un insieme di `Dictionary<TKey, TElement>` basato su una sorgente l `Dictionary<TKey, TElement> IEnumerable<T>` .

```
IEnumerable<User> users = GetUsers();
Dictionary<int, User> usersById = users.ToDictionary(x => x.Id);
```

In questo esempio, il singolo argomento passato a `ToDictionary` è di tipo `Func<TSource, TKey>` , che restituisce la chiave per ciascun elemento.

Questo è un modo conciso per eseguire la seguente operazione:

```
Dictionary<int, User> usersById = new Dictionary<int User>();
foreach (User u in users)
{
  usersById.Add(u.Id, u);
}
```

È anche possibile passare un secondo parametro al metodo `ToDictionary` , che è di tipo `Func<TSource, TElement>` e restituisce il `Value` da aggiungere per ogni voce.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, string> userNamesById = users.ToDictionary(x => x.Id, x => x.Name);
```

È anche possibile specificare `IComparer` che viene utilizzato per confrontare i valori chiave. Questo può essere utile quando la chiave è una stringa e vuoi che corrisponda a maiuscole e minuscole.

```
IEnumerable<User> users = GetUsers();
Dictionary<string, User> usersByCaseInsenstiveName = users.ToDictionary(x => x.Name,
StringComparer.InvariantCultureIgnoreCase);

var user1 = usersByCaseInsenstiveName["john"];
var user2 = usersByCaseInsenstiveName["JOHN"];
user1 == user2; // Returns true
```

Nota: il metodo `ToDictionary` richiede che tutte le chiavi siano univoche, non ci devono essere chiavi duplicate. Se ci sono, allora viene generata un'eccezione: `ArgumentException: An item with the same key has already been added.` Se hai uno scenario in cui sai che avrai più elementi con la stessa chiave, allora preferisci utilizzare `ToLookup` .

Aggregato

`Aggregate` Applica una funzione di accumulatore su una sequenza.

```
int[] intList = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = intList.Aggregate((prevSum, current) => prevSum + current);
// sum = 55
```

- Al primo passo `prevSum = 1`
- Al secondo `prevSum = prevSum(at the first step) + 2`
- Al passo i-esimo `prevSum = prevSum(at the (i-1) step) + i-th element of the array` passo `prevSum = prevSum(at the (i-1) step) + i-th element of the array`

```
string[] stringList = { "Hello", "World", "!" };
string joinedString = stringList.Aggregate((prev, current) => prev + " " + current);
// joinedString = "Hello World !"
```

Un secondo overload di `Aggregate` riceve anche un parametro `seed` che è il valore dell'accumulatore iniziale. Questo può essere usato per calcolare più condizioni su una collezione senza iterarlo più di una volta.

```
List<int> items = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Per la raccolta di `items` che vogliamo calcolare

1. Il totale `.Count`
2. La quantità di numeri pari
3. Colleziona ogni singolo oggetto

Usando `Aggregate` può essere fatto in questo modo:

```
var result = items.Aggregate(new { Total = 0, Even = 0, FourthItems = new List<int>() },
                (accumelative,item) =>
                new {
                    Total = accumelative.Total + 1,
                    Even = accumelative.Even + (item % 2 == 0 ? 1 : 0),
                    FourthItems = (accumelative.Total + 1)%4 == 0 ?
                        new List<int>(accumelative.FourthItems) { item } :
                        accumelative.FourthItems
                });
// Result:
// Total = 12
// Even = 6
// FourthItems = [4, 8, 12]
```

Si noti che l'utilizzo di un tipo anonimo come seed deve istanziare un nuovo oggetto ogni elemento perché le proprietà sono di sola lettura. Usando una classe personalizzata si può semplicemente assegnare l'informazione e non è necessario alcun `new` (solo quando si fornisce il parametro iniziale del `seed`

Definizione di una variabile all'interno di una query Linq (let keyword)

Per definire una variabile all'interno di un'espressione linq, è possibile utilizzare la parola chiave **let** . Questo di solito è fatto per archiviare i risultati di sottoquery intermedie, ad esempio:

```
 int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

 var aboveAverages = from number in numbers
                     let average = numbers.Average()
                     let nSquared = Math.Pow(number,2)
                     where nSquared > average
                     select number;

 Console.WriteLine("The average of the numbers is {0}.", numbers.Average());

 foreach (int n in aboveAverages)
 {
   Console.WriteLine("Query result includes number {0} with square of {1}.", n,
Math.Pow(n,2));
 }
```

Produzione:

La media dei numeri è 4.5.
Il risultato della query include il numero 3 con il quadrato di 9.
Il risultato della query include il numero 4 con il quadrato di 16.
Il risultato della query include il numero 5 con il quadrato di 25.

Il risultato della query include il numero 6 con il quadrato di 36.
Il risultato della query include il numero 7 con il quadrato di 49.
Il risultato della query include il numero 8 con un quadrato di 64.
Il risultato della query include il numero 9 con il quadrato di 81.

Visualizza la demo

SkipWhile

`SkipWhile()` viene utilizzato per escludere elementi fino alla prima non corrispondenza (questo potrebbe essere controintuitivo per la maggior parte)

```
int[] list = { 42, 42, 6, 6, 6, 42 };
var result = list.SkipWhile(i => i == 42);
// Result: 6, 6, 6, 42
```

DefaultIfEmpty

DefaultIfEmpty viene utilizzato per restituire un elemento predefinito se la sequenza non contiene elementi. Questo elemento può essere il valore predefinito del tipo o un'istanza definita dall'utente di quel tipo. Esempio:

```
var chars = new List<string>() { "a", "b", "c", "d" };

chars.DefaultIfEmpty("N/A").FirstOrDefault(); // returns "a";

chars.Where(str => str.Length > 1)
     .DefaultIfEmpty("N/A").FirstOrDefault(); // return "N/A"

chars.Where(str => str.Length > 1)
        .DefaultIfEmpty().First(); // returns null;
```

Utilizzo in unione sinistra :

Con `DefaultIfEmpty` il tradizionale Linq Join può restituire un oggetto predefinito se non è stata trovata alcuna corrispondenza. Così agendo come un join sinistro di SQL. Esempio:

```
var leftSequence = new List<int>() { 99, 100, 5, 20, 102, 105 };
var rightSequence = new List<char>() { 'a', 'b', 'c', 'i', 'd' };

var numbersAsChars = from l in leftSequence
                     join r in rightSequence
                     on l equals (int)r into leftJoin
                     from result in leftJoin.DefaultIfEmpty('?')
                     select new
                     {
                         Number = l,
                         Character = result
                     };

foreach(var item in numbersAsChars)
{
```

```
    Console.WriteLine("Num = {0} ** Char = {1}", item.Number, item.Character);
}

ouput:

Num = 99         Char = c
Num = 100        Char = d
Num = 5          Char = ?
Num = 20         Char = ?
Num = 102        Char = ?
Num = 105        Char = i
```

Nel caso in cui venga utilizzato un `DefaultIfEmpty` (senza specificare un valore predefinito) e ciò comporterà l'assenza di elementi corrispondenti nella sequenza corretta, è necessario assicurarsi che l'oggetto non sia `null` prima di accedere alle sue proprietà. Altrimenti risulterà in una `NullReferenceException` . Esempio:

```
var leftSequence = new List<int> { 1, 2, 5 };
var rightSequence = new List<dynamic>()
    {
        new { Value = 1 },
        new { Value = 2 },
        new { Value = 3 },
        new { Value = 4 },
    };

var numbersAsChars = (from l in leftSequence
                      join r in rightSequence
                      on l equals r.Value into leftJoin
                      from result in leftJoin.DefaultIfEmpty()
                      select new
                      {
                          Left = l,
                          // 5 will not have a matching object in the right so result
                          // will be equal to null.
                          // To avoid an error use:
                          //    -  C# 6.0 or above - ?.
                          //    -  Under           - result == null ? 0 : result.Value
                          Right = result?.Value
                      }).ToList();
```

SequenceEqual

`SequenceEqual` viene utilizzato per confrontare due sequenze I `IEnumerable<T>` Enumerabili l'una con l'altra.

```
int[] a = new int[] {1, 2, 3};
int[] b = new int[] {1, 2, 3};
int[] c = new int[] {1, 3, 2};

bool returnsTrue = a.SequenceEqual(b);
bool returnsFalse = a.SequenceEqual(c);
```

Count e LongCount

`Count` restituisce il numero di elementi in un oggetto `IEnumerable<T>` . `Count` espone anche un parametro di predicato opzionale che consente di filtrare gli elementi che si desidera contare.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

int n = array.Count(); // returns the number of elements in the array
int x = array.Count(i => i > 2); // returns the number of elements in the array greater than 2
```

`LongCount` funziona allo stesso modo di `Count` ma ha un tipo restituito di `long` e viene utilizzato per il conteggio `IEnumerable<T>` sequenze `IEnumerable<T>` che sono più lunghe di `int.MaxValue`

```
int[] array = GetLargeArray();

long n = array.LongCount(); // returns the number of elements in the array
long x = array.LongCount(i => i > 100); // returns the number of elements in the array greater
than 100
```

Costruzione incrementale di una query

Poiché LINQ utilizza l' **esecuzione posticipata** , possiamo avere un oggetto query che in realtà non contiene i valori, ma restituirà i valori quando valutato. Possiamo quindi creare dinamicamente la query in base al nostro flusso di controllo e valutarla una volta che abbiamo finito:

```
IEnumerable<VehicleModel> BuildQuery(int vehicleType, SearchModel search, int start = 1, int
count = -1) {
    IEnumerable<VehicleModel> query = _entities.Vehicles
        .Where(x => x.Active && x.Type == vehicleType)
        .Select(x => new VehicleModel {
            Id = v.Id,
            Year = v.Year,
            Class = v.Class,
            Make = v.Make,
            Model = v.Model,
            Cylinders = v.Cylinders ?? 0
        });
```

Possiamo applicare condizionatamente filtri:

```
    if (!search.Years.Contains("all", StringComparer.OrdinalIgnoreCase))
        query = query.Where(v => search.Years.Contains(v.Year));

    if (!search.Makes.Contains("all", StringComparer.OrdinalIgnoreCase)) {
        query = query.Where(v => search.Makes.Contains(v.Make));
    }

    if (!search.Models.Contains("all", StringComparer.OrdinalIgnoreCase)) {
        query = query.Where(v => search.Models.Contains(v.Model));
    }

    if (!search.Cylinders.Equals("all", StringComparer.OrdinalIgnoreCase)) {
        decimal minCylinders = 0;
        decimal maxCylinders = 0;
        switch (search.Cylinders) {
            case "2-4":
```

```
                maxCylinders = 4;
                break;
            case "5-6":
                minCylinders = 5;
                maxCylinders = 6;
                break;
            case "8":
                minCylinders = 8;
                maxCylinders = 8;
                break;
            case "10+":
                minCylinders = 10;
                break;
        }
        if (minCylinders > 0) {
            query = query.Where(v => v.Cylinders >= minCylinders);
        }
        if (maxCylinders > 0) {
            query = query.Where(v => v.Cylinders <= maxCylinders);
        }
    }
```

Possiamo aggiungere un ordinamento alla query in base a una condizione:

```
    switch (search.SortingColumn.ToLower()) {
        case "make_model":
            query = query.OrderBy(v => v.Make).ThenBy(v => v.Model);
            break;
        case "year":
            query = query.OrderBy(v => v.Year);
            break;
        case "engine_size":
            query = query.OrderBy(v => v.EngineSize).ThenBy(v => v.Cylinders);
            break;
        default:
            query = query.OrderBy(v => v.Year); //The default sorting.
    }
```

La nostra query può essere definita per iniziare da un punto specifico:

```
    query = query.Skip(start - 1);
```

e definito per restituire un numero specifico di record:

```
    if (count > -1) {
        query = query.Take(count);
    }
    return query;
}
```

Una volta ottenuto l'oggetto query, possiamo valutare i risultati con un ciclo `foreach` o uno dei metodi LINQ che restituisce un insieme di valori, come `ToList` o `ToArray` :

```
SearchModel sm;
```

```
// populate the search model here
// ...

List<VehicleModel> list = BuildQuery(5, sm).ToList();
```

Cerniera lampo

Il metodo di estensione `Zip` agisce su due raccolte. Associa ogni elemento delle due serie in base alla posizione. Con un'istanza di `Func` , utilizziamo `Zip` per gestire gli elementi delle due raccolte C # in coppie. Se le serie differiscono per dimensioni, gli elementi extra delle serie più grandi verranno ignorati.

Per fare un esempio dal libro "C # in a Nutshell",

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip(words, (n, w) => n + "=" + w);
```

Produzione:

3 = tre
5 = cinque
7 = sette

Visualizza la demo

GroupJoin con variabile range esterno

```
Customer[] customers = Customers.ToArray();
Purchase[] purchases = Purchases.ToArray();

var groupJoinQuery =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select new
    {
        CustName = c.Name,
        custPurchases
    };
```

ElementAt e ElementAtOrDefault

`ElementAt` restituirà l'oggetto all'indice `n` . Se `n` non rientra nell'intervallo dell'enumerabile, genera un'eccezione `ArgumentOutOfRangeException` .

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAt(2);  // 3
numbers.ElementAt(10); // throws ArgumentOutOfRangeException
```

`ElementAtOrDefault` restituirà l'elemento all'indice `n` . Se `n` non è compreso nell'intervallo

enumerabile, restituisce un `default(T)` .

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAtOrDefault(2);  // 3
numbers.ElementAtOrDefault(10); // 0 = default(int)
```

Sia `ElementAt` che `ElementAtOrDefault` sono ottimizzati per quando la sorgente è un `IList<T>` e in questi casi verrà utilizzata l'indicizzazione normale.

Si noti che per `ElementAt` , se l'indice fornito è maggiore della dimensione del `IList<T>` , l'elenco dovrebbe (ma tecnicamente non è garantito) generare una `ArgumentOutOfRangeException` .

Quantificatori di Linq

Le operazioni Quantifier restituiscono un valore booleano se alcuni o tutti gli elementi di una sequenza soddisfano una condizione. In questo articolo vedremo alcuni comuni scenari LINQ su oggetti in cui possiamo utilizzare questi operatori. Esistono 3 operazioni Quantifiers che possono essere utilizzate in LINQ:

`All` - utilizzato per determinare se tutti gli elementi di una sequenza soddisfano una condizione. Per esempio:

```
int[] array = { 10, 20, 30 };

// Are all elements >= 10? YES
array.All(element => element >= 10);

// Are all elements >= 20? NO
array.All(element => element >= 20);

// Are all elements < 40? YES
array.All(element => element < 40);
```

`Any` : utilizzato per determinare se alcuni elementi di una sequenza soddisfano una condizione. Per esempio:

```
int[] query=new int[] { 2, 3, 4 }
query.Any (n => n == 3);
```

`Contains` : utilizzato per determinare se una sequenza contiene un elemento specificato. Per esempio:

```
//for int array
int[] query =new int[] { 1,2,3 };
query.Contains(1);

//for string array
string[] query={"Tom","grey"};
query.Contains("Tom");

//for a string
var stringValue="hello";
```

```
stringValue.Contains("h");
```

Unire più sequenze

Considerare le entità `Customer` , `Purchase` e `PurchaseItem` come segue:

```
public class Customer
{
   public string Id { get; set } // A unique Id that identifies customer
   public string Name  {get; set; }
}

public class Purchase
{
   public string Id { get; set }
   public string CustomerId {get; set; }
   public string Description { get; set; }
}

public class PurchaseItem
{
   public string Id { get; set }
   public string PurchaseId {get; set; }
   public string Detail { get; set; }
}
```

Prendi in considerazione i seguenti dati di esempio per le entità sopra indicate:

```
var customers = new List<Customer>()
 {
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer1"
    },

    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer2"
    }
 };

 var purchases = new List<Purchase>()
 {
     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[0].Id,
         Description = "Customer1-Purchase1"
     },

     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[0].Id,
         Description = "Customer1-Purchase2"
     },

     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[1].Id,
```

```
        Description = "Customer2-Purchase1"
    },

    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase2"
    }
 };

var purchaseItems = new List<PurchaseItem>()
{
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[0].Id,
        Detail = "Purchase1-PurchaseItem1"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem1"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem2"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[3].Id,
        Detail = "Purchase3-PurchaseItem1"
    }
};
```

Ora, considera la seguente query linq:

```
var result = from c in customers
             join p in purchases on c.Id equals p.CustomerId           // first join
             join pi in purchaseItems on p.Id equals pi.PurchaseId     // second join
             select new
             {
                c.Name, p.Description, pi.Detail
             };
```

Per generare il risultato della query precedente:

```
foreach(var resultItem in result)
{
    Console.WriteLine($"{resultItem.Name}, {resultItem.Description}, {resultItem.Detail}");
}
```

L'output della query sarebbe:

> Customer1, Customer1-Purchase1, Purchase1-PurchaseItem1

Customer1, Customer1-Purchase2, Purchase2-PurchaseItem1

Customer1, Customer1-Purchase2, Purchase2-PurchaseItem2

Customer2, Customer2-Purchase2, Purchase3-PurchaseItem1

Live Demo su .NET Fiddle

Partecipare a più chiavi

```
  PropertyInfo[] stringProps = typeof (string).GetProperties();//string properties
  PropertyInfo[] builderProps = typeof(StringBuilder).GetProperties();//stringbuilder
properties

    var query =
        from s in stringProps
        join b in builderProps
            on new { s.Name, s.PropertyType } equals new { b.Name, b.PropertyType }
        select new
        {
            s.Name,
            s.PropertyType,
            StringToken = s.MetadataToken,
            StringBuilderToken = b.MetadataToken
        };
```

Si noti che i tipi anonimi in `join` sopra devono contenere le stesse proprietà poiché gli oggetti sono considerati uguali solo se tutte le loro proprietà sono uguali. Altrimenti la query non verrà compilata.

Seleziona con Func selettore - Utilizzare per ottenere il ranking degli elementi

Sui sovraccarichi dei metodi di estensione `Select` passa anche l' `index` dell'elemento corrente nella raccolta `select` . Questi sono alcuni usi di esso.

Ottieni il "numero di riga" degli articoli

```
var rowNumbers = collection.OrderBy(item => item.Property1)
                           .ThenBy(item => item.Property2)
                           .ThenByDescending(item => item.Property3)
                           .Select((item, index) => new { Item = item, RowNumber = index })
                           .ToList();
```

Ottieni il rango di un oggetto *all'interno del* suo gruppo

```
var rankInGroup = collection.GroupBy(item => item.Property1)
                            .OrderBy(group => group.Key)
                            .SelectMany(group => group.OrderBy(item => item.Property2)
                                                      .ThenByDescending(item => item.Property3)
                                                      .Select((item, index) => new
                                                      {
                                                          Item = item,
                                                          RankInGroup = index
                                                      })).ToList();
```

Ottieni la classifica dei gruppi (noto anche in Oracle come dense_rank)

```
var rankOfBelongingGroup = collection.GroupBy(item => item.Property1)
                            .OrderBy(group => group.Key)
                            .Select((group, index) => new
                            {
                                Items = group,
                                Rank = index
                            })
                            .SelectMany(v => v.Items, (s, i) => new
                            {
                                Item = i,
                                DenseRank = s.Rank
                            }).ToList();
```

Per testare questo è possibile utilizzare:

```
public class SomeObject
{
    public int Property1 { get; set; }
    public int Property2 { get; set; }
    public int Property3 { get; set; }

    public override string ToString()
    {
        return string.Join(", ", Property1, Property2, Property3);
    }
}
```

E i dati:

```
List<SomeObject> collection = new List<SomeObject>
{
    new SomeObject { Property1 = 1, Property2 = 1, Property3 = 1},
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 2},
    new SomeObject { Property1 = 2, Property2 = 1, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 3, Property3 = 1}
};
```

TakeWhile

`TakeWhile` restituisce elementi da una sequenza purché la condizione sia vera

```
int[] list = { 1, 10, 40, 50, 44, 70, 4 };
var result = list.TakeWhile(item => item < 50).ToList();
// result = { 1, 10, 40 }
```

Somma

Il metodo di estensione `Enumerable.Sum` calcola la somma dei valori numerici.

Nel caso in cui gli elementi della raccolta siano essi stessi numeri, è possibile calcolare la somma direttamente.

```
int[] numbers = new int[] { 1, 4, 6 };
Console.WriteLine( numbers.Sum() ); //outputs 11
```

Nel caso in cui il tipo degli elementi sia di tipo complesso, è possibile utilizzare un'espressione lambda per specificare il valore da calcolare:

```
var totalMonthlySalary = employees.Sum( employee => employee.MonthlySalary );
```

Il metodo di estensione somma può calcolare con i seguenti tipi:

- Int32
- Int64
- singolo
- Doppio
- Decimale

Nel caso in cui la tua raccolta contenga tipi annullabili, puoi utilizzare l'operatore null-coalescing per impostare un valore predefinito per elementi null:

```
int?[] numbers = new int?[] { 1, null, 6 };
Console.WriteLine( numbers.Sum( number => number ?? 0 ) ); //outputs 7
```

ToLookup

> ToLookup restituisce una struttura dati che consente l'indicizzazione. È un metodo di estensione. Produce un'istanza di ILookup che può essere indicizzata o enumerata utilizzando un ciclo foreach. Le voci sono combinate in raggruppamenti per ogni chiave. - dotnetperls

```
string[] array = { "one", "two", "three" };
//create lookup using string length as key
var lookup = array.ToLookup(item => item.Length);

//join the values whose lengths are 3
Console.WriteLine(string.Join(",",lookup[3]));
//output: one,two
```

Un altro esempio:

```
int[] array = { 1,2,3,4,5,6,7,8 };
//generate lookup for odd even numbers (keys will be 0 and 1)
var lookup = array.ToLookup(item => item % 2);

//print even numbers after joining
Console.WriteLine(string.Join(",",lookup[0]));
//output: 2,4,6,8
```

```
//print odd numbers after joining
Console.WriteLine(string.Join(",",lookup[1]));
//output: 1,3,5,7
```

Costruisci i tuoi operatori Linq per IEnumerable

Una delle grandi cose di Linq è che è così facile da estendere. Hai solo bisogno di creare un metodo di estensione il cui argomento è `IEnumerable<T>` .

```
public namespace MyNamespace
{
    public static class LinqExtensions
    {
        public static IEnumerable<List<T>> Batch<T>(this IEnumerable<T> source, int batchSize)
        {
            var batch = new List<T>();
            foreach (T item in source)
            {
                batch.Add(item);
                if (batch.Count == batchSize)
                {
                    yield return batch;
                    batch = new List<T>();
                }
            }
            if (batch.Count > 0)
                yield return batch;
        }
    }
}
```

Questo esempio suddivide gli elementi in un oggetto `IEnumerable<T>` in elenchi di dimensioni fisse, l'ultimo elenco contenente il resto degli elementi. Si noti come l'oggetto a cui viene applicato il metodo di estensione viene passato in (argomento `source`) come argomento iniziale utilizzando la parola chiave `this` . Quindi la parola chiave `yield` viene utilizzata per generare l'elemento successivo nell'output `IEnumerable<T>` prima di continuare con l'esecuzione da quel punto (vedere la parola chiave yield).

Questo esempio verrebbe utilizzato nel tuo codice in questo modo:

```
//using MyNamespace;
var items = new List<int> { 2, 3, 4, 5, 6 };
foreach (List<int> sublist in items.Batch(3))
{
    // do something
}
```

Nel primo ciclo, il sottolista sarebbe `{2, 3, 4}` e il secondo `{5, 6}` .

I metodi LinQ personalizzati possono anche essere combinati con i metodi LinQ standard. per esempio:

```
//using MyNamespace;
```

```
var result = Enumerable.Range(0, 13)         // generate a list
                       .Where(x => x%2 == 0) // filter the list or do something other
                       .Batch(3)             // call our extension method
                       .ToList()             // call other standard methods
```

Questa query restituirà numeri pari raggruppati in batch con una dimensione di 3: `{0, 2, 4}, {6, 8, 10}, {12}`

Ricorda che hai bisogno di `using MyNamespace;` linea per poter accedere al metodo di estensione.

Usando SelectMany invece di cicli annidati

Dato 2 elenchi

```
var list1 = new List<string> { "a", "b", "c" };
var list2 = new List<string> { "1", "2", "3", "4" };
```

se vuoi esportare tutte le permutazioni puoi usare loop annidati come

```
var result = new List<string>();
foreach (var s1 in list1)
    foreach (var s2 in list2)
        result.Add($"{s1}{s2}");
```

Usando SelectMany puoi fare la stessa operazione di

```
var result = list1.SelectMany(x => list2.Select(y => $"{x}{y}", x, y)).ToList();
```

Any and First (OrDefault): best practice

Non spiegherò cosa fa `Any` e `FirstOrDefault` perché ci sono già due buoni esempi su di loro. Vedere Any and First, FirstOrDefault, Last, LastOrDefault, Single e SingleOrDefault per ulteriori informazioni.

Un modello che vedo spesso nel codice che **dovrebbe essere evitato** è

```
if (myEnumerable.Any(t=>t.Foo == "Bob"))
{
    var myFoo = myEnumerable.First(t=>t.Foo == "Bob");
    //Do stuff
}
```

Potrebbe essere scritto in modo più efficiente come questo

```
var myFoo = myEnumerable.FirstOrDefault(t=>t.Foo == "Bob");
if (myFoo != null)
{
    //Do stuff
}
```

Utilizzando il secondo esempio, la raccolta viene ricercata una sola volta e restituisce lo stesso

risultato della prima. La stessa idea può essere applicata a `Single` .

GroupBy Sum e Count

Prendiamo una classe di esempio:

```
public class Transaction
{
    public string Category { get; set; }
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
}
```

Ora, consideriamo un elenco di transazioni:

```
var transactions = new List<Transaction>
{
   new Transaction { Category = "Saving Account", Amount = 56, Date =
DateTime.Today.AddDays(1) },
   new Transaction { Category = "Saving Account", Amount = 10, Date = DateTime.Today.AddDays(-
10) },
   new Transaction { Category = "Credit Card", Amount = 15, Date = DateTime.Today.AddDays(1)
},
   new Transaction { Category = "Credit Card", Amount = 56, Date = DateTime.Today },
   new Transaction { Category = "Current Account", Amount = 100, Date =
DateTime.Today.AddDays(5) },
};
```

Se si desidera calcolare la somma saggia della categoria di importo e contare, è possibile utilizzare GroupBy come segue:

```
var summaryApproach1 = transactions.GroupBy(t => t.Category)
                           .Select(t => new
                           {
                               Category = t.Key,
                               Count = t.Count(),
                               Amount = t.Sum(ta => ta.Amount),
                           }).ToList();

Console.WriteLine("-- Summary: Approach 1 --");
summaryApproach1.ForEach(
            row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

In alternativa, puoi farlo in un solo passaggio:

```
var summaryApproach2 = transactions.GroupBy(t => t.Category, (key, t) =>
{
        var transactionArray = t as Transaction[] ?? t.ToArray();
        return new
        {
            Category = key,
            Count = transactionArray.Length,
            Amount = transactionArray.Sum(ta => ta.Amount),
        };
```

```
}).ToList();

Console.WriteLine("-- Summary: Approach 2 --");
summaryApproach2.ForEach(
row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

L'output per entrambe le query sopra sarebbe lo stesso:

> Categoria: Conto di risparmio, Quantità: 66, Numero: 2
>
> Categoria: Carta di credito, quantità: 71, Count: 2
>
> Categoria: conto corrente, importo: 100, numero: 1

Demo dal vivo in .NET Fiddle

Inverso

- Inverte l'ordine degli elementi in una sequenza.
- Se non ci sono elementi lancia una `ArgumentNullException: source is null.`

Esempio:

```
// Create an array.
int[] array = { 1, 2, 3, 4 };                          //Output:
// Call reverse extension method on the array.         //4
var reverse = array.Reverse();                         //3
// Write contents of array to screen.                  //2
foreach (int value in reverse)                         //1
    Console.WriteLine(value);
```

Esempio di codice in tempo reale

Ricorda che `Reverse()` può funzionare in modo diverso a seconda dell'ordine della catena delle istruzioni LINQ.

```
        //Create List of chars
        List<int> integerlist = new List<int>() { 1, 2, 3, 4, 5, 6 };

        //Reversing the list then taking the two first elements
        IEnumerable<int> reverseFirst = integerlist.Reverse<int>().Take(2);

        //Taking 2 elements and then reversing only thos two
        IEnumerable<int> reverseLast = integerlist.Take(2).Reverse();

        //reverseFirst output: 6, 5
        //reverseLast output:  2, 1
```

Esempio di codice in tempo reale

Reverse () funziona con il buffering di tutto quindi lo attraversa all'indietro, non è molto efficiente, ma nemmeno OrderBy da quella prospettiva.

In LINQ-to-Objects, ci sono operazioni di buffering (Reverse, OrderBy, GroupBy, ecc.) E operazioni di non buffering (Where, Take, Skip, ecc.).

Esempio: estensione non bufferizzata inversa

```
public static IEnumerable<T> Reverse<T>(this IList<T> list) {
    for (int i = list.Count - 1; i >= 0; i--)
        yield return list[i];
}
```

Esempio di codice in tempo reale

Questo metodo può incontrare problemi se si modifica l'elenco durante l'iterazione.

Enumerazione dell'enumerabile

L'interfaccia IEnumerable <T> è l'interfaccia di base per tutti gli enumeratori generici ed è una parte essenziale per la comprensione di LINQ. Nel suo nucleo, rappresenta la sequenza.

Questa interfaccia sottostante è ereditata da tutte le raccolte generiche, come Collection <T> , Array , List <T> , Dictionary <TKey, TValue> Class e HashSet <T> .

Oltre a rappresentare la sequenza, qualsiasi classe che eredita da IEnumerable <T> deve fornire un IEnumerator <T>. L'enumeratore espone l'iteratore per l'enumerabile, e queste due interfacce e idee interconnesse sono la fonte del detto "enumerare l'enumerabile".

"Enumerare l'enumerabile" è una frase importante. L'enumerabile è semplicemente una struttura su come iterare, non contiene oggetti materializzati. Ad esempio, durante l'ordinamento, un enumerable può contenere i criteri del campo da ordinare, ma l'utilizzo di `.OrderBy()` in sé restituirà un IEnumerable <T> che sa solo *come* ordinare. L'uso di una chiamata che materializzerà gli oggetti, come in iterare l'insieme, è noto come enumerazione (ad esempio `.ToList()`). Il processo di enumerazione utilizzerà la definizione enumerabile di *come,* al fine di muoversi attraverso le serie e restituire gli oggetti rilevanti (in ordine, filtrato, proiettato, ecc.).

Solo una volta che l'enumerabile è stato enumerato, ciò provoca la materializzazione degli oggetti, ovvero quando le metriche come la complessità temporale (quanto tempo occorre prendere in relazione alle dimensioni della serie) e la complessità spaziale (quanto spazio dovrebbe usare in relazione alle dimensioni della serie) possono essere misurato

La creazione della propria classe che eredita da IEnumerable <T> può essere un po 'complicata a seconda delle serie sottostanti che devono essere enumerabili. In generale, è meglio utilizzare una delle raccolte generiche esistenti. Detto questo, è anche possibile ereditare dall'interfaccia IEnumerable <T> senza avere una matrice definita come struttura sottostante.

Ad esempio, utilizzando la serie di Fibonacci come sequenza sottostante. Si noti che la chiamata a `Where` semplicemente costruisce un oggetto `IEnumerable` , e non è fino a quando una chiamata per enumerare quella enumerabile è fatta che tutti i valori sono materializzati.

```
void Main()
```

```
{
    Fibonacci Fibo = new Fibonacci();
    IEnumerable<long> quadrillionplus = Fibo.Where(i => i > 1000000000000);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(quadrillionplus.Take(2).Sum());
    Console.WriteLine(quadrillionplus.Skip(2).First());

    IEnumerable<long> fibMod612 = Fibo.OrderBy(i => i % 612);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(fibMod612.First());//smallest divisible by 612
}

public class Fibonacci : IEnumerable<long>
{
    private int max = 90;

    //Enumerator called typically from foreach
    public IEnumerator GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }

    //Enumerable called typically from linq
    IEnumerator<long> IEnumerable<long>.GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }
}
```

Produzione

```
Enumerable built
Enumerating the Enumerable
4052739537881
Enumerating the Enumerable
4052739537881
Enumerable built
Enumerating the Enumerable
14930352
```

La forza nel secondo set (il fibMod612) è che anche se abbiamo fatto la chiamata per ordinare l'intero set di numeri di Fibonacci, poiché è stato preso solo un valore usando `.First()` la complessità temporale era O (n) come solo 1 valore doveva essere confrontato durante l'esecuzione dell'algoritmo di ordinazione. Questo perché il nostro enumeratore ha chiesto solo 1 valore, e quindi non è stato necessario materializzare l'intero enumerabile. Se avessimo usato `.Take(5)` anziché `.First()` l'enumeratore avrebbe richiesto 5 valori e al massimo 5 valori avrebbero

dovuto essere materializzati. Rispetto alla necessità di ordinare un intero set *e quindi* prendere i primi 5 valori, il principio di risparmiare un sacco di tempo e spazio di esecuzione.

Ordinato da

Ordina una collezione in base a un valore specificato.

Quando il valore è un **numero intero** , **doppio** o **float** inizia con il *valore minimo* , il che significa che si ottengono prima i valori negativi, che zero e afterwords i valori positivi (vedere Esempio 1).

Quando ordini per **char,** il metodo confronta i *valori ascii* dei caratteri per ordinare la raccolta (vedi Esempio 2).

Quando ordinate le **stringhe,** il metodo OrderBy le confronta confrontandole con CultureInfo, ma normalmente iniziano con la *prima lettera* dell'alfabeto (a, b, c ...).

Questo tipo di ordine è chiamato ascendente, se lo vuoi viceversa hai bisogno di scendere (vedi OrderByDescending).

Esempio 1:

```
int[] numbers = {2, 1, 0, -1, -2};
IEnumerable<int> ascending = numbers.OrderBy(x => x);
// returns {-2, -1, 0, 1, 2}
```

Esempio 2:

```
 char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z',
'Z'};
 IEnumerable<char> ascending = letters.OrderBy(x => x);
 // returns { ' ', '!', '+', '1', '9', '?', 'A', 'B', 'Y', 'Z', '[', 'a', 'b', 'y', 'z', '{' }
```

Esempio:

```
class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var youngestPerson = people.OrderBy(x => x.Age).First();
var name = youngestPerson.Name; // Bob
```

OrderByDescending

Ordina una collezione in base a un valore specificato.

Quando il valore è un **numero intero** , **doppio** o **float** inizia con il *valore massimo* , il che significa che si ottengono prima i valori positivi, che zero e afterwords i valori negativi (vedere Esempio 1).

Quando ordini per **char,** il metodo confronta i *valori ascii* dei caratteri per ordinare la raccolta (vedi Esempio 2).

Quando si ordinano le **stringhe,** il metodo OrderBy li confronta dando un'occhiata al loro CultureInfo ma normalmente iniziando con l' *ultima lettera* dell'alfabeto (z, y, x, ...).

Questo tipo di ordine è chiamato discendente, se lo vuoi viceversa hai bisogno di salire (vedi OrderBy).

Esempio 1:

```
int[] numbers = {-2, -1, 0, 1, 2};
IEnumerable<int> descending = numbers.OrderByDescending(x => x);
// returns {2, 1, 0, -1, -2}
```

Esempio 2:

```
char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z',
'Z'};
IEnumerable<char> descending = letters.OrderByDescending(x => x);
// returns { '{', 'z', 'y', 'b', 'a', '[', 'Z', 'Y', 'B', 'A', '?', '9', '1', '+', '!', ' ' }
```

Esempio 3:

```
class Person
{
   public  string Name { get; set; }
   public  int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var oldestPerson = people.OrderByDescending(x => x.Age).First();
var name = oldestPerson.Name; // Carol
```

concat

Unisce due raccolte (senza rimuovere duplicati)

```
List<int> foo = new List<int> { 1, 2, 3 };
List<int> bar = new List<int> { 3, 4, 5 };

// Through Enumerable static class
var result = Enumerable.Concat(foo, bar).ToList(); // 1,2,3,3,4,5

// Through extension method
```

```
var result = foo.Concat(bar).ToList(); // 1,2,3,3,4,5
```

contiene

MSDN:

> Determina se una sequenza contiene un elemento specificato utilizzando un `IEqualityComparer<T>` specificato `IEqualityComparer<T>`

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
var result1 = numbers.Contains(4); // true
var result2 = numbers.Contains(8); // false

List<int> secondNumberCollection = new List<int> { 4, 5, 6, 7 };
// Note that can use the Intersect method in this case
var result3 = secondNumberCollection.Where(item => numbers.Contains(item)); // will be true
only for 4,5
```

Utilizzando un oggetto definito dall'utente:

```
public class Person
{
   public string Name { get; set; }
}

List<Person> objects = new List<Person>
{
    new Person { Name = "Nikki"},
    new Person { Name = "Gilad"},
    new Person { Name = "Phil"},
    new Person { Name = "John"}
};

//Using the Person's Equals method - override Equals() and GetHashCode() - otherwise it
//will compare by reference and result will be false
var result4 = objects.Contains(new Person { Name = "Phil" }); // true
```

Utilizzo del sovraccarico `Enumerable.Contains(value, comparer)` :

```
public class Compare : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name;
    }
    public int GetHashCode(Person codeh)
    {
        return codeh.Name.GetHashCode();
    }
}

var result5 = objects.Contains(new Person { Name = "Phil" }, new Compare()); // true
```

Un uso intelligente di `Contains` sarebbe quello di sostituire più clausole `if` in una chiamata `Contains` .

Quindi, invece di fare questo:

```
if(status == 1 || status == 3 || status == 4)
{
    //Do some business operation
}
else
{
    //Do something else
}
```

Fai questo:

```
if(new int[] {1, 3, 4 }.Contains(status)
{
    //Do some business operaion
}
else
{
    //Do something else
}
```

Leggi Query LINQ online: https://riptutorial.com/it/csharp/topic/68/query-linq

Capitolo 130: Reactive Extensions (Rx)

Examples

Osservazione dell'evento TextChanged su un controllo TextBox

Un osservabile viene creato dall'evento TextChanged del TextBox. Inoltre, qualsiasi input viene selezionato solo se è diverso dall'ultimo input e se non vi è stato input entro 0,5 secondi. L'output in questo esempio viene inviato alla console.

```
Observable
     .FromEventPattern(textBoxInput, "TextChanged")
     .Select(s => ((TextBox) s.Sender).Text)
     .Throttle(TimeSpan.FromSeconds(0.5))
     .DistinctUntilChanged()
     .Subscribe(text => Console.WriteLine(text));
```

Streaming dei dati dal database con Observable

Si supponga di avere un metodo che restituisce `IEnumerable<T>` , fe

```
private IEnumerable<T> GetData()
{
    try
    {
        // return results from database
    }
    catch(Exception exception)
    {
        throw;
    }
}
```

Crea un Osservabile e avvia un metodo in modo asincrono. `SelectMany` appiattisce la raccolta e l'abbonamento viene attivato ogni 200 elementi tramite `Buffer` .

```
int bufferSize = 200;

Observable
    .Start(() => GetData())
    .SelectMany(s => s)
    .Buffer(bufferSize)
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe(items =>
    {
        Console.WriteLine("Loaded {0} elements", items.Count);

        // do something on the UI like incrementing a ProgressBar
    },
    () => Console.WriteLine("Completed loading"));
```

Leggi Reactive Extensions (Rx) online: https://riptutorial.com/it/csharp/topic/5770/reactive-extensions--rx-

Capitolo 131: Regex Parsing

Sintassi

- `new Regex(pattern);` *// Crea una nuova istanza con un modello definito.*
- `Regex.Match(input);` *// Avvia la ricerca e restituisce la corrispondenza.*
- `Regex.Matches(input);` *// Avvia la ricerca e restituisce un MatchCollection*

Parametri

Nome	Dettagli
Modello	Il modello di `string` che deve essere utilizzato per la ricerca. Per maggiori informazioni: msdn
RegexOptions *[Opzionale]*	Le opzioni comuni qui sono `Singleline` e `Multiline` . Stanno cambiando il comportamento degli elementi modello come il punto (.) Che non coprirà una `NewLine` (\ n) in `Multiline-Mode` ma in `SingleLine-Mode` . Comportamento predefinito: msdn
Timeout *[Opzionale]*	Laddove i modelli diventano più complessi, la ricerca può consumare più tempo. Questo è il timeout passato per la ricerca, così come noto dalla programmazione di rete.

Osservazioni

Hai bisogno di usare

```
using System.Text.RegularExpressions;
```

Bello avere

- Puoi testare i tuoi modelli online senza la necessità di compilare la tua soluzione per ottenere risultati qui: Fai clic su di me
- Regex101 Esempio: clic me

Soprattutto i principianti tendono a sovraffaticare i loro compiti con regex perché si sente potente e nel posto giusto per ricerche più complesse basate sul testo. Questo è il punto in cui le persone cercano di analizzare i documenti xml con espressioni regolari senza nemmeno chiedere a se stessi se ci potrebbe essere una classe già finita per questo compito come `XmlDocument` .

Regex dovrebbe essere l'ultima arma per raccogliere la complessità di nuovo. Almeno non dimenticarti di mettere in atto uno sforzo per cercare la `right way` prima di scrivere 20 linee di

modelli.

Examples

Partita singola

```
using System.Text.RegularExpressions;
```

```
string pattern = ":(.*?):";
string lookup = "--:text in here:--";

// Instanciate your regex object and pass a pattern to it
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));
// Get the match from your regex-object
Match mLookup = rgxLookup.Match(lookup);

// The group-index 0 always covers the full pattern.
// Matches inside parentheses will be accessed through the index 1 and above.
string found = mLookup.Groups[1].Value;
```

Risultato:

```
found = "text in here"
```

Più partite

```
using System.Text.RegularExpressions;
```

```
List<string> found = new List<string>();
string pattern = ":(.*?):";
string lookup = "--:text in here:--:another one:-:third one:---!123:fourth:";

// Instanciate your regex object and pass a pattern to it
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));
MatchCollection mLookup = rgxLookup.Matches(lookup);

foreach(Match match in mLookup)
{
    found.Add(match.Groups[1].Value);
}
```

Risultato:

```
found = new List<string>() { "text in here", "another one", "third one", "fourth" }
```

Leggi Regex Parsing online: https://riptutorial.com/it/csharp/topic/3774/regex-parsing

Capitolo 132: Rendere sicuro un thread variabile

Examples

Controllo dell'accesso a una variabile in un ciclo Parallel.Per

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main( string[] args )
    {
        object sync = new object();
        int sum = 0;
        Parallel.For( 1, 1000, ( i ) => {
            lock( sync ) sum = sum + i; // lock is necessary

            // As a practical matter, ensure this `parallel for` executes
            // on multiple threads by simulating a lengthy operation.
            Thread.Sleep( 1 );
        } );
        Console.WriteLine( "Correct answer should be 499500.  sum is: {0}", sum );
    }
}
```

Non è sufficiente fare `sum = sum + i` senza il lock perché l'operazione read-modify-write non è atomica. Un thread sovrascriverà tutte le modifiche esterne alla `sum` che si verificano dopo aver letto il valore corrente della `sum` , ma prima che esso memorizzi il valore modificato di `sum + i` nuovo in `sum` .

Leggi Rendere sicuro un thread variabile online:
https://riptutorial.com/it/csharp/topic/4140/rendere-sicuro-un-thread-variabile

Capitolo 133: ricorsione

Osservazioni

Si noti che l'utilizzo della ricorsione può avere un impatto grave sul codice, poiché ogni chiamata di funzione ricorsiva verrà aggiunta allo stack. Se ci sono troppe chiamate questo potrebbe portare a **un'eccezione StackOverflow** . La maggior parte delle "funzioni ricorsive naturali" possono essere scritti come `for` , `while` o `foreach` costrutto di ciclo, e pur non guardare in modo **elegante** o **intelligente** sarà più efficiente.

Pensaci sempre due volte e usa ricorsione con attenzione - sappi perché lo usi:

- la ricorsione dovrebbe essere usata quando si sa che il numero di chiamate ricorsive non è *eccessivo*
    - mezzi *eccessivi* , dipende da quanta memoria è disponibile
- la ricorsione viene utilizzata perché è una versione del codice più chiara e più pulita, è più leggibile di una funzione iterativa o basata su loop. Spesso questo è il caso perché dà un codice più pulito e più compatto (ovvero meno linee di codice).
    - ma attenzione, può essere meno efficiente! Ad esempio nella ricorsione di Fibonacci, per calcolare l' *ennesimo* numero nella sequenza, il tempo di calcolo crescerà esponenzialmente!

Se vuoi più teoria, leggi:

- https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/recursion2.html
- https://en.wikipedia.org/wiki/Recursion#In_computer_science

Examples

Descrivere ricorsivamente una struttura dell'oggetto

La ricorsione è quando un metodo si chiama da solo. Preferibilmente lo farà finché non verrà soddisfatta una condizione specifica e quindi uscirà normalmente dal metodo, ritornando al punto da cui è stato chiamato il metodo. In caso contrario, potrebbe verificarsi un'eccezione di overflow dello stack dovuta a troppe chiamate ricorsive.

```
/// <summary>
/// Create an object structure the code can recursively describe
/// </summary>
public class Root
{
    public string Name { get; set; }
    public ChildOne Child { get; set; }
}
public class ChildOne
{
    public string ChildOneName { get; set; }
    public ChildTwo Child { get; set; }
```

```
}
public class ChildTwo
{
    public string ChildTwoName { get; set; }
}
/// <summary>
/// The console application with the recursive function DescribeTypeOfObject
/// </summary>
public class Program
{
    static void Main(string[] args)
    {
        // point A, we call the function with type 'Root'
        DescribeTypeOfObject(typeof(Root));
        Console.WriteLine("Press a key to exit");
        Console.ReadKey();
    }

    static void DescribeTypeOfObject(Type type)
    {
        // get all properties of this type
        Console.WriteLine($"Describing type {type.Name}");
        PropertyInfo[] propertyInfos = type.GetProperties();
        foreach (PropertyInfo pi in propertyInfos)
        {
            Console.WriteLine($"Has property {pi.Name} of type {pi.PropertyType.Name}");
            // is a custom class type? describe it too
            if (pi.PropertyType.IsClass && !pi.PropertyType.FullName.StartsWith("System."))
            {
                // point B, we call the function type this property
                DescribeTypeOfObject(pi.PropertyType);
            }
        }
        // done with all properties
        // we return to the point where we were called
        // point A for the first call
        // point B for all properties of type custom class
    }
}
```

Ricorsione in inglese semplice

La ricorsione può essere definita come:

> Un metodo che chiama se stesso finché non viene soddisfatta una condizione specifica.

Un esempio eccellente e semplice di ricorsione è un metodo che otterrà il fattoriale di un dato numero:

```
public int Factorial(int number)
{
    return number == 0 ? 1 : n * Factorial(number - 1);
}
```

In questo metodo, possiamo vedere che il metodo prenderà un argomento, `number` .

Passo dopo passo:

Dato l'esempio, eseguendo `Factorial(4)`

1. Il `number (4) == 1` ?
2. No? return `4 * Factorial(number-1)` (3)
3. Poiché il metodo viene chiamato ancora una volta, ripete ora il primo passo utilizzando `Factorial(3)` come nuovo argomento.
4. Questo continua finché non viene eseguito `Factorial(1)` e `number (1) == 1` restituisce 1.
5. Complessivamente, il calcolo "crea" `4 * 3 * 2 * 1` e infine restituisce 24.

La chiave per comprendere la ricorsione è che il metodo chiama una *nuova istanza* di se stesso. Dopo il ritorno, l'esecuzione dell'istanza chiamante continua.

Utilizzo della ricorsione per ottenere la struttura della directory

Uno degli usi della ricorsione consiste nel navigare attraverso una struttura gerarchica dei dati, come un albero di directory del file system, senza sapere quanti livelli ha l'albero o il numero di oggetti su ciascun livello. In questo esempio, vedrai come usare la ricorsione su un albero di directory per trovare tutte le sottodirectory di una directory specificata e stampare l'intero albero sulla console.

```
internal class Program
{
    internal const int RootLevel = 0;
    internal const char Tab = '\t';

    internal static void Main()
    {
        Console.WriteLine("Enter the path of the root directory:");
        var rootDirectorypath = Console.ReadLine();

        Console.WriteLine(
            $"Getting directory tree of '{rootDirectorypath}'");

        PrintDirectoryTree(rootDirectorypath);
        Console.WriteLine("Press 'Enter' to quit...");
        Console.ReadLine();
    }

    internal static void PrintDirectoryTree(string rootDirectoryPath)
    {
        try
        {
            if (!Directory.Exists(rootDirectoryPath))
            {
                throw new DirectoryNotFoundException(
                    $"Directory '{rootDirectoryPath}' not found.");
            }

            var rootDirectory = new DirectoryInfo(rootDirectoryPath);
            PrintDirectoryTree(rootDirectory, RootLevel);
        }
        catch (DirectoryNotFoundException e)
        {
```

```
            Console.WriteLine(e.Message);
        }
    }

    private static void PrintDirectoryTree(
        DirectoryInfo directory, int currentLevel)
    {
        var indentation = string.Empty;
        for (var i = RootLevel; i < currentLevel; i++)
        {
            indentation += Tab;
        }

        Console.WriteLine($"{indentation}-{directory.Name}");
        var nextLevel = currentLevel + 1;
        try
        {
            foreach (var subDirectory in directory.GetDirectories())
            {
                PrintDirectoryTree(subDirectory, nextLevel);
            }
        }
        catch (UnauthorizedAccessException e)
        {
            Console.WriteLine($"{indentation}-{e.Message}");
        }
    }
}
```

Questo codice è un po 'più complicato del minimo indispensabile per completare questa attività, poiché include il controllo delle eccezioni per gestire eventuali problemi con l'ottenimento delle directory. Di seguito troverai una suddivisione del codice in segmenti più piccoli con le spiegazioni di ciascuno.

`Main` :

Il metodo principale accetta un input da un utente come una stringa, che deve essere utilizzato come percorso della directory principale. Quindi chiama il metodo `PrintDirectoryTree` con questa stringa come parametro.

`PrintDirectoryTree(string)` :

Questo è il primo di due metodi che gestiscono la stampa dell'albero della directory effettiva. Questo metodo prende una stringa che rappresenta il percorso della directory root come parametro. Controlla se il percorso è una directory effettiva e, in caso contrario, genera una `DirectoryNotFoundException` che viene quindi gestita nel blocco catch. Se il percorso è una directory reale, una `DirectoryInfo` oggetto `rootDirectory` viene creato dal percorso, e la seconda `PrintDirectoryTree` metodo viene chiamato con il `rootDirectory` oggetto e `RootLevel` , che è un numero intero costante con un valore di zero.

`PrintDirectoryTree(DirectoryInfo, int)` :

Questo secondo metodo gestisce il peso del lavoro. Prende un `DirectoryInfo` e un intero come parametri. `DirectoryInfo` è la directory corrente e il numero intero è la profondità della directory

relativa alla radice. Per facilità di lettura, l'output è indentato per ogni livello in profondità nella directory corrente, in modo che l'output assomigli a questo:

```
-Root
    -Child 1
    -Child 2
        -Grandchild 2.1
    -Child 3
```

Una volta stampata la directory corrente, vengono recuperate le sottodirectory e questo metodo viene quindi chiamato su ciascuno di essi con un valore del livello di profondità superiore a quello corrente. Quella parte è la ricorsione: il metodo che si autodefinisce. Il programma verrà eseguito in questo modo finché non avrà visitato tutte le directory dell'albero. Quando ha raggiunto una directory senza subdirectory, il metodo restituirà automaticamente.

Questo metodo rileva anche una `UnauthorizedAccessException` , che viene generata se una delle sottodirectory della directory corrente è protetta dal sistema. Il messaggio di errore viene stampato al livello di indentazione corrente per coerenza.

Il metodo seguente fornisce un approccio più di base a questo problema:

```
internal static void PrintDirectoryTree(string directoryName)
{
    try
    {
        if (!Directory.Exists(directoryName)) return;
        Console.WriteLine(directoryName);
        foreach (var d in Directory.GetDirectories(directoryName))
        {
            PrintDirectoryTree(d);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Questo non include lo specifico controllo degli errori o la formattazione dell'output del primo approccio, ma fa effettivamente la stessa cosa. Poiché utilizza solo stringhe anziché `DirectoryInfo` , non può fornire l'accesso ad altre proprietà di directory come le autorizzazioni.

Sequenza di Fibonacci

Puoi calcolare un numero nella sequenza di Fibonacci usando la ricorsione.

Seguendo la teoria matematica di F (n) = F (n-2) + F (n-1), per ogni i> 0,

```
// Returns the i'th Fibonacci number
public int fib(int i) {
    if(i <= 2) {
        // Base case of the recursive function.
        // i is either 1 or 2, whose associated Fibonacci sequence numbers are 1 and 1.
```

```
        return 1;
    }
    // Recursive case. Return the sum of the two previous Fibonacci numbers.
    // This works because the definition of the Fibonacci sequence specifies
    // that the sum of two adjacent elements equals the next element.
    return  fib(i - 2) + fib(i - 1);

}

fib(10); // Returns 55
```

Calcolo fattoriale

Il fattoriale di un numero (indicato con!, Come per esempio 9!) È la moltiplicazione di quel numero con il fattoriale di uno inferiore. Quindi, per esempio, 9! = 9 x 8! = 9 x 8 x 7! = 9 x 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1.

Quindi nel codice che diventa, usando la ricorsione:

```
long Factorial(long x)
{
    if (x < 1)
    {
        throw new OutOfRangeException("Factorial can only be used with positive numbers.");
    }

    if (x == 1)
    {
        return 1;
    } else {
        return x * Factorial(x - 1);
    }
}
```

Calcolo PowerOf

Il calcolo della potenza di un dato numero può essere eseguito in modo ricorsivo. Dato un numero base `n` ed esponente `e` , dobbiamo assicurarci di dividere il problema in blocchi diminuendo l'esponente `e` .

Esempio teorico:

- $2^2 = 2x2$
- $2^3 = 2x2x2$ o, $2^3 = 2^2 \times 2$
  Qui sta il segreto del nostro algoritmo ricorsivo (vedi il codice sotto). Si tratta di prendere il problema e separarlo in blocchi più piccoli e più semplici da risolvere.
- **Gli appunti**
  - quando il numero di base è 0, dobbiamo essere consapevoli di restituire 0 come $0^3 = 0 \times 0 \times 0$
  - quando l'esponente è 0, dobbiamo essere consapevoli di restituire sempre 1, in quanto questa è una regola matematica.

Esempio di codice:

```
public int CalcPowerOf(int b, int e) {
    if (b == 0) { return 0; } // when base is 0, it doesn't matter, it will always return 0
    if (e == 0) { return 1; } // math rule, exponent 0 always returns 1
    return b * CalcPowerOf(b, e - 1); // actual recursive logic, where we split the problem,
aka: 2³ = 2 * 2² etc..
}
```

Test in xUnit per verificare la logica:
Sebbene ciò non sia necessario, è sempre consigliabile scrivere test per verificare la logica.
Includo quelli qui scritti nel framework xUnit .

```
    [Theory]
    [MemberData(nameof(PowerOfTestData))]
    public void PowerOfTest(int @base, int exponent, int expected) {
        Assert.Equal(expected, CalcPowerOf(@base, exponent));
    }

    public static IEnumerable<object[]> PowerOfTestData() {
        yield return new object[] { 0, 0, 0 };
        yield return new object[] { 0, 1, 0 };
        yield return new object[] { 2, 0, 1 };
        yield return new object[] { 2, 1, 2 };
        yield return new object[] { 2, 2, 4 };
        yield return new object[] { 5, 2, 25 };
        yield return new object[] { 5, 3, 125 };
        yield return new object[] { 5, 4, 625 };
}
```

Leggi ricorsione online: https://riptutorial.com/it/csharp/topic/2470/ricorsione

Capitolo 134: Riflessione

introduzione

Reflection è un meccanismo in linguaggio C # per accedere alle proprietà degli oggetti dinamici in runtime. In genere, reflection viene utilizzato per recuperare le informazioni sul tipo di oggetto dinamico e sui valori degli attributi dell'oggetto. Nell'applicazione REST, ad esempio, è possibile utilizzare la riflessione per scorrere l'oggetto di risposta serializzato.

Nota: in base alle linee guida MS, il codice critico delle prestazioni dovrebbe evitare la riflessione. Vedere https://msdn.microsoft.com/en-us/library/ff647790.aspx

Osservazioni

Reflection consente al codice di accedere alle informazioni su assiemi, moduli e tipi in fase di esecuzione (esecuzione del programma). Questo può quindi essere ulteriormente utilizzato per creare, modificare o accedere dinamicamente ai tipi. I tipi includono proprietà, metodi, campi e attributi.

Ulteriori letture:

Riflessione (C #)

Riflessione in .Net Framework

Examples

Ottieni un System.Type

Per un'istanza di un tipo:

```
var theString = "hello";
var theType = theString.GetType();
```

Dal tipo stesso:

```
var theType = typeof(string);
```

Ottieni i membri di un tipo

```
using System;
using System.Reflection;
using System.Linq;

public class Program
{
```

```
  public static void Main()
  {
    var members = typeof(object)
                    .GetMembers(BindingFlags.Public |
                                BindingFlags.Static |
                                BindingFlags.Instance);

    foreach (var member in members)
    {
      bool inherited = member.DeclaringType.Equals( typeof(object).Name );
      Console.WriteLine($"{member.Name} is a {member.MemberType}, " +
                        $"it has {(inherited ? "":"not")} been inherited.");
    }
  }
}
```

Uscita (*vedi nota sull'ordine di uscita più in basso*):

```
GetType is a Method, it has not been inherited.
GetHashCode is a Method, it has not been inherited.
ToString is a Method, it has not been inherited.
Equals is a Method, it has not been inherited.
Equals is a Method, it has not been inherited.
ReferenceEquals is a Method, it has not been inherited.
.ctor is a Constructor, it has not been inherited.
```

Possiamo anche utilizzare `GetMembers()` senza passare `BindingFlags` . Ciò restituirà *tutti* i membri pubblici di quel tipo specifico.

Una cosa da notare che `GetMembers` non restituisce i membri in alcun ordine particolare, quindi non fare mai affidamento sull'ordine che `GetMembers` ti restituisce.

Visualizza la demo

Ottieni un metodo e invocalo

Ottieni il metodo di istanza e invocalo

```
using System;

public class Program
{
    public static void Main()
    {
        var theString = "hello";
        var method = theString
                    .GetType()
                    .GetMethod("Substring",
                               new[] {typeof(int), typeof(int)}); //The types of the method
arguments
         var result = method.Invoke(theString, new object[] {0, 4});
         Console.WriteLine(result);
    }
}
```

Produzione:

inferno

Visualizza la demo

Ottieni il metodo statico e invocalo

D'altra parte, se il metodo è statico, non è necessaria un'istanza per chiamarlo.

```
var method = typeof(Math).GetMethod("Exp");
var result = method.Invoke(null, new object[] {2});//Pass null as the first argument (no need
for an instance)
Console.WriteLine(result); //You'll get e^2
```

Produzione:

7,38905609893065

Visualizza la demo

Ottenere e impostare le proprietà

Utilizzo di base:

```
PropertyInfo prop = myInstance.GetType().GetProperty("myProperty");
// get the value myInstance.myProperty
object value = prop.GetValue(myInstance);

int newValue = 1;
// set the value myInstance.myProperty to newValue
prop.setValue(myInstance, newValue);
```

L'impostazione delle proprietà di sola lettura implementate automaticamente può essere effettuata tramite il relativo campo di supporto (in .NET Framework il nome del campo di supporto è "k__BackingField"):

```
// get backing field info
FieldInfo fieldInfo = myInstance.GetType()
    .GetField("<myProperty>k__BackingField", BindingFlags.Instance | BindingFlags.NonPublic);

int newValue = 1;
// set the value of myInstance.myProperty backing field to newValue
fieldInfo.SetValue(myInstance, newValue);
```

Attributi personalizzati

Trova proprietà con un attributo personalizzato - `MyAttribute`

```
var props = t.GetProperties(BindingFlags.NonPublic | BindingFlags.Public |
            BindingFlags.Instance).Where(
            prop => Attribute.IsDefined(prop, typeof(MyAttribute)));
```

Trova tutti gli attributi personalizzati su una determinata proprietà

```
var attributes = typeof(t).GetProperty("Name").GetCustomAttributes(false);
```

Enumerare tutte le classi con attributo personalizzato - `MyAttribute`

```
static IEnumerable<Type> GetTypesWithAttribute(Assembly assembly) {
    foreach(Type type in assembly.GetTypes()) {
        if (type.GetCustomAttributes(typeof(MyAttribute), true).Length > 0) {
            yield return type;
        }
    }
}
```

Leggi il valore di un attributo personalizzato in fase di esecuzione

```
public static class AttributeExtensions
{

        /// <summary>
        /// Returns the value of a member attribute for any member in a class.
        ///     (a member is a Field, Property, Method, etc...)
        /// <remarks>
        /// If there is more than one member of the same name in the class, it will return the
first one (this applies to overloaded methods)
        /// </remarks>
        /// <example>
        /// Read System.ComponentModel Description Attribute from method 'MyMethodName' in
class 'MyClass':
        ///     var Attribute = typeof(MyClass).GetAttribute("MyMethodName",
(DescriptionAttribute d) => d.Description);
        /// </example>
        /// <param name="type">The class that contains the member as a type</param>
        /// <param name="MemberName">Name of the member in the class</param>
        /// <param name="valueSelector">Attribute type and property to get (will return first
instance if there are multiple attributes of the same type)</param>
        /// <param name="inherit">true to search this member's inheritance chain to find the
attributes; otherwise, false. This parameter is ignored for properties and events</param>
        /// </summary>
        public static TValue GetAttribute<TAttribute, TValue>(this Type type, string
MemberName, Func<TAttribute, TValue> valueSelector, bool inherit = false) where TAttribute :
Attribute
        {
            var att =
type.GetMember(MemberName).FirstOrDefault().GetCustomAttributes(typeof(TAttribute),
inherit).FirstOrDefault() as TAttribute;
            if (att != null)
            {
                return valueSelector(att);
            }
            return default(TValue);
        }
    }
```

uso

```
//Read System.ComponentModel Description Attribute from method 'MyMethodName' in class
```

```
'MyClass'
var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d) =>
d.Description);
```

Andare in loop attraverso tutte le proprietà di una classe

```
Type type = obj.GetType();
//To restrict return properties. If all properties are required don't provide flag.
BindingFlags flags = BindingFlags.Public | BindingFlags.Instance;
PropertyInfo[] properties = type.GetProperties(flags);

foreach (PropertyInfo property in properties)
{
    Console.WriteLine("Name: " + property.Name + ", Value: " + property.GetValue(obj, null));
}
```

Determinazione di argomenti generici di istanze di tipi generici

Se si dispone di un'istanza di un tipo generico ma per qualche motivo non si conosce il tipo specifico, è possibile determinare gli argomenti generici che sono stati utilizzati per creare questa istanza.

Supponiamo che qualcuno abbia creato un'istanza di `List<T>` come tale e lo trasmetta a un metodo:

```
var myList = new List<int>();
ShowGenericArguments(myList);
```

dove `ShowGenericArguments` ha questa firma:

```
public void ShowGenericArguments(object o)
```

quindi al momento della compilazione non hai idea di quali argomenti generici sono stati usati per creare `o` . Reflection fornisce molti metodi per ispezionare i tipi generici. Inizialmente, possiamo determinare se il tipo di `o` è del tutto generico:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;

    Type t = o.GetType();
    if (!t.IsGenericType) return;
    ...
```

`Type.IsGenericType` restituisce `true` se il tipo è di tipo generico e `false` caso contrario.

Ma questo non è tutto ciò che vogliamo sapere. `List<>` è un tipo generico. Ma vogliamo solo esaminare le istanze di specifici tipi *generici costruiti* . Un tipo generico costruito è ad esempio un `List<int>` che ha un *argomento di* tipo specifico per tutti i suoi *parametri* generici.

La classe `Type` fornisce altre due proprietà, `IsConstructedGenericType` e `IsGenericTypeDefinition` , per

distinguere questi tipi generici costruiti da definizioni di tipi generici:

```
typeof(List<>).IsGenericType // true
typeof(List<>).IsGenericTypeDefinition // true
typeof(List<>).IsConstructedGenericType// false

typeof(List<int>).IsGenericType // true
typeof(List<int>).IsGenericTypeDefinition // false
typeof(List<int>).IsConstructedGenericType// true
```

Per enumerare gli argomenti generici di un'istanza, possiamo utilizzare il metodo `GetGenericArguments()` che restituisce una matrice `Type` contenente gli argomenti di tipo generico:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;
    Type t = o.GetType();
    if (!t.IsConstructedGenericType) return;

    foreach(Type genericTypeArgument in t.GetGenericArguments())
        Console.WriteLine(genericTypeArgument.Name);
}
```

Quindi la chiamata dall'alto (`ShowGenericArguments(myList)`) produce questo risultato:

```
Int32
```

Ottieni un metodo generico e invocalo

Diciamo che hai lezione con metodi generici. E devi chiamare le sue funzioni con la riflessione.

```
public class Sample
{
    public void GenericMethod<T>()
    {
        // ...
    }

    public static void StaticMethod<T>()
    {
        //...
    }
}
```

Diciamo che vogliamo chiamare GenericMethod con tipo string.

```
Sample sample = new Sample();//or you can get an instance via reflection

MethodInfo method = typeof(Sample).GetMethod("GenericMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(sample, null);//Since there are no arguments, we are passing null
```

Per il metodo statico non hai bisogno di un'istanza. Quindi anche il primo argomento sarà nullo.

```
MethodInfo method = typeof(Sample).GetMethod("StaticMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(null, null);
```

Crea un'istanza di un tipo generico e invoca il suo metodo

```
var baseType = typeof(List<>);
var genericType = baseType.MakeGenericType(typeof(String));
var instance = Activator.CreateInstance(genericType);
var method = genericType.GetMethod("GetHashCode");
var result = method.Invoke(instance, new object[] { });
```

Istanze di istanziazione che implementano un'interfaccia (es. Attivazione di plugin)

Se si desidera che l'applicazione per supportare un sistema di plug-in, ad esempio per caricare i plugin da assemblee situati in `plugins` cartella:

```
interface IPlugin
{
    string PluginDescription { get; }
    void DoWork();
}
```

Questa classe si troverà in una dll separata

```
class HelloPlugin : IPlugin
{
    public string PluginDescription => "A plugin that says Hello";
    public void DoWork()
    {
        Console.WriteLine("Hello");
    }
}
```

Il caricatore di plugin dell'applicazione troverà i file dll, otterrà tutti i tipi in quegli assembly che implementano `IPlugin` e ne creerà istanze.

```
    public IEnumerable<IPlugin> InstantiatePlugins(string directory)
    {
        var pluginAssemblyNames = Directory.GetFiles(directory, "*.addin.dll").Select(name =>
new FileInfo(name).FullName).ToArray();
        //load the assemblies into the current AppDomain, so we can instantiate the types
later
        foreach (var fileName in pluginAssemblyNames)
            AppDomain.CurrentDomain.Load(File.ReadAllBytes(fileName));
        var assemblies = pluginAssemblyNames.Select(System.Reflection.Assembly.LoadFile);
        var typesInAssembly = assemblies.SelectMany(asm => asm.GetTypes());
        var pluginTypes = typesInAssembly.Where(type => typeof
(IPlugin).IsAssignableFrom(type));
        return pluginTypes.Select(Activator.CreateInstance).Cast<IPlugin>();
    }
```

Creazione di un'istanza di un tipo

Il modo più semplice è usare la classe `Activator` .

Tuttavia, anche se le prestazioni di `Activator` sono state migliorate da .NET 3.5, l'uso di `Activator.CreateInstance()` è un'opzione errata a volte, a causa di prestazioni (relativamente) basse: Test 1 , Test 2 , Test 3 ...

Con classe `Activator`

```
Type type = typeof(BigInteger);
object result = Activator.CreateInstance(type); //Requires parameterless constructor.
Console.WriteLine(result); //Output: 0
result = Activator.CreateInstance(type, 123); //Requires a constructor which can receive an
'int' compatible argument.
Console.WriteLine(result); //Output: 123
```

È possibile passare un array di oggetti a `Activator.CreateInstance` se si dispone di più di un parametro.

```
// With a constructor such as MyClass(int, int, string)
Activator.CreateInstance(typeof(MyClass), new object[] { 1, 2, "Hello World" });

Type type = typeof(someObject);
var instance = Activator.CreateInstance(type);
```

Per un tipo generico

Il metodo `MakeGenericType` trasforma un tipo generico aperto (come `List<>`) in un tipo concreto (come `List<string>`) applicando argomenti tipo ad esso.

```
// generic List with no parameters
Type openType = typeof(List<>);

// To create a List<string>
Type[] tArgs = { typeof(string) };
Type target = openType.MakeGenericType(tArgs);

// Create an instance - Activator.CreateInstance will call the default constructor.
// This is equivalent to calling new List<string>().
List<string> result = (List<string>)Activator.CreateInstance(target);
```

La sintassi di `List<>` non è consentita al di fuori di un `typeof` espressione.

Senza classe `Activator`

Usando la `new` parola chiave (lo farà per i costruttori senza parametri)

```
T GetInstance<T>() where T : new()
{
    T instance = new T();
    return instance;
}
```

Utilizzo del metodo Invoke

```
// Get the instance of the desired constructor (here it takes a string as a parameter).
ConstructorInfo c = typeof(T).GetConstructor(new[] { typeof(string) });
// Don't forget to check if such constructor exists
if (c == null)
    throw new InvalidOperationException(string.Format("A constructor for type '{0}' was not
found.", typeof(T)));
T instance = (T)c.Invoke(new object[] { "test" });
```

Uso degli alberi di espressione

Gli alberi di espressione rappresentano il codice in una struttura dati ad albero, in cui ogni nodo è un'espressione. Come spiega MSDN :

> L'espressione è una sequenza di uno o più operandi e zero o più operatori che possono essere valutati in un singolo valore, oggetto, metodo o spazio dei nomi. Le espressioni possono essere costituite da un valore letterale, un'invocazione di metodo, un operatore e i suoi operandi o un nome semplice. I nomi semplici possono essere il nome di una variabile, un membro del tipo, un parametro del metodo, uno spazio dei nomi o un tipo.

```
public class GenericFactory<TKey, TType>
    {
       private readonly Dictionary<TKey, Func<object[], TType>> _registeredTypes; //
dictionary, that holds constructor functions.
       private object _locker = new object(); // object for locking dictionary, to guarantee
thread safety

        public GenericFactory()
        {
            _registeredTypes = new Dictionary<TKey, Func<object[], TType>>();
        }

        /// <summary>
        /// Find and register suitable constructor for type
        /// </summary>
        /// <typeparam name="TType"></typeparam>
        /// <param name="key">Key for this constructor</param>
        /// <param name="parameters">Parameters</param>
        public void Register(TKey key, params Type[] parameters)
        {
            ConstructorInfo ci = typeof(TType).GetConstructor(BindingFlags.Public |
BindingFlags.Instance, null, CallingConventions.HasThis, parameters, new ParameterModifier[] {
}); // Get the instance of ctor.
            if (ci == null)
                throw new InvalidOperationException(string.Format("Constructor for type '{0}'
was not found.", typeof(TType)));

            Func<object[], TType> ctor;
```

```
            lock (_locker)
            {
                if (!_registeredTypes.TryGetValue(key, out ctor)) // check if such ctor
already been registered
                {
                    var pExp = Expression.Parameter(typeof(object[]), "arguments"); // create
parameter Expression
                    var ctorParams = ci.GetParameters(); // get parameter info from
constructor

                    var argExpressions = new Expression[ctorParams.Length]; // array that will
contains parameter expessions
                    for (var i = 0; i < parameters.Length; i++)
                    {

                        var indexedAcccess = Expression.ArrayIndex(pExp,
Expression.Constant(i));

                        if (!parameters[i].IsClass && !parameters[i].IsInterface) // check if
parameter is a value type
                        {
                            var localVariable = Expression.Variable(parameters[i],
"localVariable"); // if so - we should create local variable that will store paraameter value

                            var block = Expression.Block(new[] { localVariable },
                                    Expression.IfThenElse(Expression.Equal(indexedAcccess,
Expression.Constant(null)),
                                        Expression.Assign(localVariable,
Expression.Default(parameters[i])),
                                        Expression.Assign(localVariable,
Expression.Convert(indexedAcccess, parameters[i]))
                                    ),
                                    localVariable
                                );

                            argExpressions[i] = block;

                        }
                        else
                            argExpressions[i] = Expression.Convert(indexedAcccess,
parameters[i]);
                    }
                    var newExpr = Expression.New(ci, argExpressions); // create expression
that represents call to specified ctor with the specified arguments.

                    _registeredTypes.Add(key, Expression.Lambda(newExpr, new[] { pExp
}).Compile() as Func<object[], TType>); // compile expression to create delegate, and add
fucntion to dictionary
                }
            }
        }

        /// <summary>
        /// Returns instance of registered type by key.
        /// </summary>
        /// <typeparam name="TType"></typeparam>
        /// <param name="key"></param>
        /// <param name="args"></param>
        /// <returns></returns>
        public TType Create(TKey key, params object[] args)
```

```
        {
            Func<object[], TType> foo;
            if (_registeredTypes.TryGetValue(key, out foo))
            {
                return (TType)foo(args);
            }

            throw new ArgumentException("No type registered for this key.");
        }
    }
```

Potrebbe essere usato in questo modo:

```
 public class TestClass
 {
        public TestClass(string parameter)
        {
            Console.Write(parameter);
        }
 }


public void TestMethod()
{
       var factory = new GenericFactory<string, TestClass>();
       factory.Register("key", typeof(string));
       TestClass newInstance = factory.Create("key", "testParameter");
}
```

Utilizzo di FormatterServices.GetUninitializedObject

```
T instance = (T)FormatterServices.GetUninitializedObject(typeof(T));
```

In caso di utilizzo di costruttori e di inizializzatori di campo `FormatterServices.GetUninitializedObject` non verranno chiamati. È pensato per essere utilizzato nei serializzatori e nei motori remoti

Ottieni un tipo per nome con namespace

Per fare questo è necessario un riferimento all'assembly che contiene il tipo. Se hai un altro tipo disponibile che sai è nello stesso assembly di quello che desideri, puoi farlo:

```
typeof(KnownType).Assembly.GetType(typeName);
```

- dove `typeName` è il nome del tipo che stai cercando (incluso lo spazio dei nomi), e `KnownType` è il tipo che conosci è nello stesso assembly.

Meno efficiente ma più generale è il seguente:

```
Type t = null;
foreach (Assembly ass in AppDomain.CurrentDomain.GetAssemblies())
{
    if (ass.FullName.StartsWith("System."))
```

```
        continue;
    t = ass.GetType(typeName);
    if (t != null)
        break;
}
```

Si noti il controllo per escludere gli assembly dello spazio dei nomi System di scansione per velocizzare la ricerca. Se il tuo tipo potrebbe effettivamente essere un tipo CLR, dovrai eliminare queste due righe.

Se ti capita di avere il nome completo del tipo completo di assembly incluso l'assembly, puoi semplicemente farlo

```
Type.GetType(fullyQualifiedName);
```

Ottieni un delegato fortemente digitato su un metodo o una proprietà tramite Reflection

Quando la prestazione è un problema, invocare un metodo tramite reflection (cioè tramite il metodo `MethodInfo.Invoke`) non è l'ideale. Tuttavia, è relativamente semplice ottenere un delegato con una forte caratterizzazione più performante utilizzando la funzione `Delegate.CreateDelegate` . La penalizzazione delle prestazioni per l'uso della riflessione viene sostenuta solo durante il processo di creazione dei delegati. Una volta che il delegato è stato creato, c'è una penalità da prestazione minima a quella per invocarlo:

```
// Get a MethodInfo for the Math.Max(int, int) method...
var maxMethod = typeof(Math).GetMethod("Max", new Type[] { typeof(int), typeof(int) });
// Now get a strongly-typed delegate for Math.Max(int, int)...
var stronglyTypedDelegate = (Func<int, int, int>)Delegate.CreateDelegate(typeof(Func<int, int,
int>), null, maxMethod);
// Invoke the Math.Max(int, int) method using the strongly-typed delegate...
Console.WriteLine("Max of 3 and 5 is: {0}", stronglyTypedDelegate(3, 5));
```

Questa tecnica può essere estesa anche alle proprietà. Se abbiamo una classe chiamata `MyClass` con una proprietà `int` nome `MyIntProperty` , il codice per ottenere un getter fortemente tipizzato sarebbe (l'esempio seguente presuppone che 'target' sia un'istanza valida di `MyClass`):

```
// Get a MethodInfo for the MyClass.MyIntProperty getter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theGetter = theProperty.GetGetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
MyClass instance...
var stronglyTypedGetter = (Func<MyClass, int>)Delegate.CreateDelegate(typeof(Func<MyClass,
int>), theGetter);
// Invoke the MyIntProperty getter against MyClass instance 'target'...
Console.WriteLine("target.MyIntProperty is: {0}", stronglyTypedGetter(target));
```

... e lo stesso può essere fatto per il setter:

```
// Get a MethodInfo for the MyClass.MyIntProperty setter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
```

```
var theSetter = theProperty.GetSetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any
MyClass instance...
var stronglyTypedSetter = (Action<MyClass, int>)Delegate.CreateDelegate(typeof(Action<MyClass,
int>), theSetter);
// Set MyIntProperty to 5...
stronglyTypedSetter(target, 5);
```

Leggi Riflessione online: https://riptutorial.com/it/csharp/topic/28/riflessione

Capitolo 135: Risoluzione di sovraccarico

Osservazioni

Il processo di risoluzione del sovraccarico è descritto nella specifica C # , sezione 7.5.3. Rilevanti sono anche le sezioni 7.5.2 (inferenza del tipo) e 7.6.5 (espressioni di invocazione).

Come funziona la risoluzione di sovraccarico verrà probabilmente modificato in C # 7. Le note di progettazione indicano che Microsoft distribuirà un nuovo sistema per determinare quale metodo è migliore (in scenari complicati).

Examples

Esempio di sovraccarico di base

Questo codice contiene un metodo sovraccarico denominato **Hello** :

```
class Example
{
    public static void Hello(int arg)
    {
        Console.WriteLine("int");
    }

    public static void Hello(double arg)
    {
        Console.WriteLine("double");
    }

    public static void Main(string[] args)
    {
        Hello(0);
        Hello(0.0);
    }
}
```

Quando viene chiamato il metodo **Main** , verrà stampato

```
int
double
```

In fase di compilazione, quando il compilatore trova il metodo call `Hello(0)` , trova tutti i metodi con il nome `Hello` . In questo caso, ne trova due. Quindi cerca di determinare quale dei metodi è *migliore* . L'algoritmo per determinare quale metodo è migliore è complesso, ma in genere si riduce a "generare il minor numero possibile di conversioni implicite".

Pertanto, nel caso di `Hello(0)` , non è necessaria alcuna conversione per il metodo `Hello(int)` ma è necessaria una conversione numerica implicita per il metodo `Hello(double)` . Quindi, il primo metodo è scelto dal compilatore.

Nel caso di `Hello(0.0)` , non esiste alcun modo per convertire implicitamente `0.0` in un `int` , quindi il metodo `Hello(int)` non è nemmeno considerato per la risoluzione di sovraccarico. Rimane solo il metodo e quindi viene scelto dal compilatore.

"params" non è espanso, se non necessario.

Il seguente programma:

```
class Program
{
    static void Method(params Object[] objects)
    {
        System.Console.WriteLine(objects.Length);
    }
    static void Method(Object a, Object b)
    {
        System.Console.WriteLine("two");
    }
    static void Main(string[] args)
    {
        object[] objectArray = new object[5];

        Method(objectArray);
        Method(objectArray, objectArray);
        Method(objectArray, objectArray, objectArray);
    }
}
```

stamperà:

```
5
two
3
```

Il `Method(objectArray)` expression call `Method(objectArray)` potrebbe essere interpretato in due modi: un singolo argomento `Object` che capita di essere un array (quindi il programma emetterebbe `1` perché sarebbe il numero di argomenti, o come una matrice di argomenti, dato nel forma normale, come se il metodo `Method` non avesse i `params` della parola chiave In queste situazioni, la forma normale, non espansa ha sempre la precedenza, quindi il programma emette `5` .

Nella seconda espressione, `Method(objectArray, objectArray)` , sono applicabili sia la forma espansa del primo metodo che il secondo metodo tradizionale. Anche in questo caso, le forme non espanse hanno la precedenza, quindi il programma ne stampa `two` .

Nella terza espressione, `Method(objectArray, objectArray, objectArray)` , l'unica opzione è utilizzare la forma espansa del primo metodo, quindi il programma stampa `3` .

Passando null come uno degli argomenti

Se hai

```
void F1(MyType1 x) {
    // do something
}

void F1(MyType2 x) {
    // do something else
}
```

e per qualche motivo devi chiamare il primo overload di `F1` ma con `x = null` , quindi fare semplicemente

```
F1(null);
```

non verrà compilato poiché la chiamata è ambigua. Per contrastare ciò che puoi fare

```
F1(null as MyType1);
```

Leggi Risoluzione di sovraccarico online: https://riptutorial.com/it/csharp/topic/77/risoluzione-di-sovraccarico

Capitolo 136: Runtime Compile

Examples

RoslynScript

`Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript` è un nuovo motore di script C #.

```
var code = "(1 + 2).ToString()";
var run = await CSharpScript.RunAsync(code, ScriptOptions.Default);
var result = (string)run.ReturnValue;
Console.WriteLine(result); //output 3
```

È possibile compilare ed eseguire dichiarazioni, variabili, metodi, classi o qualsiasi segmento di codice.

CSharpCodeProvider

`Microsoft.CSharp.CSharpCodeProvider` può essere utilizzato per compilare classi C #.

```
var code = @"
    public class Abc {
       public string Get() { return ""abc""; }
    }
";

var options = new CompilerParameters();
options.GenerateExecutable = false;
options.GenerateInMemory = false;

var provider = new CSharpCodeProvider();
var compile = provider.CompileAssemblyFromSource(options, code);

var type = compile.CompiledAssembly.GetType("Abc");
var abc = Activator.CreateInstance(type);

var method = type.GetMethod("Get");
var result = method.Invoke(abc, null);

Console.WriteLine(result); //output: abc
```

Leggi Runtime Compile online: https://riptutorial.com/it/csharp/topic/3139/runtime-compile

Capitolo 137: ruscello

Examples

Usando i flussi

Un flusso è un oggetto che fornisce un mezzo di basso livello per trasferire i dati. Loro stessi non agiscono come contenitori di dati.

I dati che trattiamo sono in forma di array di byte (`byte []`). Le funzioni di lettura e scrittura sono tutte orientate ai byte, ad esempio `WriteByte()` .

Non ci sono funzioni per gestire interi, stringhe, ecc. Ciò rende lo stream molto generico, ma meno semplice da utilizzare se, per esempio, si desidera semplicemente trasferire il testo. Gli stream possono essere particolarmente utili quando si ha a che fare con una grande quantità di dati.

Dovremo utilizzare diversi tipi di stream in base ai quali deve essere scritto / letto (ad esempio il backing store). Ad esempio, se la fonte è un file, dobbiamo usare `FileStream` :

```
string filePath = @"c:\Users\exampleuser\Documents\userinputlog.txt";
using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite))
{
    // do stuff here...

    fs.Close();
}
```

Allo stesso modo, `MemoryStream` viene utilizzato se il backing store è memoria:

```
// Read all bytes in from a file on the disk.
byte[] file = File.ReadAllBytes(“C:\\file.txt”);

// Create a memory stream from those bytes.
using (MemoryStream memory = new MemoryStream(file))
{
   // do stuff here...
}
```

Allo stesso modo, `System.Net.Sockets.NetworkStream` viene utilizzato per l'accesso alla rete.

Tutti gli stream derivano dalla classe generica `System.IO.Stream` . I dati non possono essere letti o scritti direttamente dai flussi. .NET Framework fornisce classi helper come `StreamReader` , `StreamWriter` , `BinaryReader` e `BinaryWriter` che convertono tra i tipi nativi e l'interfaccia di flusso di basso livello e trasferiscono i dati al o dallo stream per te.

La lettura e la scrittura sugli stream possono essere eseguite tramite `StreamReader` e `StreamWriter` . Si dovrebbe fare attenzione quando si chiudono questi. Per impostazione predefinita, la chiusura chiuderà anche lo stream contenuto e lo renderà inutilizzabile per ulteriori utilizzi. Questo

comportamento predefinito può essere modificato utilizzando un costruttore che ha il parametro `bool leaveOpen` e ne imposta il valore come `true` .

`StreamWriter` :

```
FileStream fs = new FileStream("sample.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
string NextLine = "This is the appended line.";
sw.Write(NextLine);
sw.Close();
//fs.Close(); There is no need to close fs. Closing sw will also close the stream it contains.
```

`StreamReader` :

```
using (var ms = new MemoryStream())
{
    StreamWriter sw = new StreamWriter(ms);
    sw.Write(123);
    //sw.Close();     This will close ms and when we try to use ms later it will cause an
exception
    sw.Flush();     //You can send the remaining data to stream. Closing will do this
automatically
    // We need to set the position to 0 in order to read
    // from the beginning.
    ms.Position = 0;
    StreamReader sr = new StreamReader(ms);
    var myStr = sr.ReadToEnd();
    sr.Close();
    ms.Close();
}
```

Poiché Classes `Stream` , `StreamReader` , `StreamWriter` , ecc. Implementano l'interfaccia `IDisposable` , possiamo chiamare il metodo `Dispose()` sugli oggetti di queste classi.

Leggi ruscello online: https://riptutorial.com/it/csharp/topic/3114/ruscello

Capitolo 138: Selezionato e deselezionato

Sintassi

- controllato (a + b) // espressione controllata
- deselezionata (a + b) // espressione non controllata
- controllato {c = a + b; c + = 5; } // blocco selezionato
- deselezionato {c = a + b; c + = 5; } // blocco deselezionato

Examples

Selezionato e deselezionato

Le istruzioni C # vengono eseguite in un contesto selezionato o non selezionato. In un contesto controllato, l'overflow aritmetico genera un'eccezione. In un contesto non controllato, l'overflow aritmetico viene ignorato e il risultato viene troncato.

```
short m = 32767;
short n = 32767;
int result1 =  checked((short)(m + n));   //will throw an OverflowException
int result2 =  unchecked((short)(m + n)); // will return -2
```

Se nessuno di questi viene specificato, il contesto predefinito si baserà su altri fattori, come le opzioni del compilatore.

Selezionato e deselezionato come ambito

Le parole chiave possono anche creare ambiti per (dis) controllare più operazioni.

```
short m = 32767;
short n = 32767;
checked
{
    int result1 = (short)(m + n); //will throw an OverflowException
}
unchecked
{
    int result2 = (short)(m + n); // will return -2
}
```

Leggi Selezionato e deselezionato online: https://riptutorial.com/it/csharp/topic/2394/selezionato-e-deselezionato

Capitolo 139: Sequenze di escape delle stringhe

Sintassi

- \ '- virgoletta singola (0x0027)
- \ "- virgolette doppie (0x0022)
- \\ - backslash (0x005C)
- \ 0 - null (0x0000)
- \ a - alert (0x0007)
- \ b - backspace (0x0008)
- \ f - form feed (0x000C)
- \ n - nuova riga (0x000A)
- \ r - ritorno a capo (0x000D)
- \ t - scheda orizzontale (0x0009)
- \ v - scheda verticale (0x000B)
- \ u0000 - \ uFFFF - Carattere Unicode
- \ x0 - \ xFFFF - Carattere Unicode (codice con lunghezza variabile)
- \ U00000000 - \ U0010FFFF - Carattere Unicode (per la generazione di surrogati)

Osservazioni

Le sequenze di escape di stringhe vengono trasformate nel carattere corrispondente in **fase** di **compilazione** . Le stringhe ordinarie che contengono stringhe all'indietro **non** vengono trasformate.

Ad esempio, le stringhe `notEscaped` e `notEscaped2` seguito non vengono trasformate in un carattere di nuova riga, ma resteranno come due caratteri diversi (`'\'` e `'n'`).

```
string escaped = "\n";
string notEscaped = "\\" + "n";
string notEscaped2 = "\\n";

Console.WriteLine(escaped.Length); // 1
Console.WriteLine(notEscaped.Length); // 2
Console.WriteLine(notEscaped2.Length); // 2
```

Examples

Sequenze di escape dei caratteri Unicode

```
string sqrt = "\u221A";      // √
string emoji = "\U0001F601"; // 
string text = "\u0022Hello World\u0022"; // "Hello World"
string variableWidth = "\x22Hello World\x22"; // "Hello World"
```

Scappare simboli speciali in caratteri letterali

Apostrophes

```
char apostrophe = '\'';
```

Barra rovesciata

```
char oneBackslash = '\\';
```

Scappare simboli speciali in stringhe letterali

Barra rovesciata

```
// The filename will be c:\myfile.txt in both cases
string filename = "c:\\myfile.txt";
string filename = @"c:\myfile.txt";
```

Il secondo esempio utilizza letteralmente una stringa letterale , che non considera la barra rovesciata come un carattere di escape.

Citazioni

```
string text = "\"Hello World!\", said the quick brown fox.";
string verbatimText = @"""Hello World!"", said the quick brown fox.";
```

Entrambe le variabili conterranno lo stesso testo.

> "Ciao mondo!", Disse la rapida volpe marrone.

newlines

I valori letterali stringa letterali possono contenere righe nuove:

```
string text = "Hello\r\nWorld!";
string verbatimText = @"Hello
World!";
```

Entrambe le variabili conterranno lo stesso testo.

Le sequenze di escape non riconosciute producono errori in fase di compilazione

I seguenti esempi non verranno compilati:

```
string s = "\c";
char c = '\c';
```

Invece, produrranno l'errore `Unrecognized escape sequence` al momento della compilazione.

Utilizzo delle sequenze di escape negli identificatori

Le sequenze di escape non sono limitate a `string` e `char` letterali.

Supponi di dover eseguire l'override di un metodo di terze parti:

```
protected abstract IEnumerable<Texte> ObtenirŒuvres();
```

e supponiamo che il carattere `Œ` non sia disponibile nella codifica dei caratteri che usi per i tuoi file sorgente C #. Sei fortunato, è consentito utilizzare gli escape del tipo `\u####` o `\U########` in ***identificatori*** nel codice. Quindi è legale scrivere:

```
protected override IEnumerable<Texte> Obtenir\u0152uvres()
{
    // ...
}
```

e il compilatore C # saprà `Œ` e `\u0152` hanno lo stesso carattere.

(Tuttavia, potrebbe essere una buona idea passare a UTF-8 o una codifica simile in grado di gestire tutti i caratteri.)

Leggi Sequenze di escape delle stringhe online: https://riptutorial.com/it/csharp/topic/39/sequenze-di-escape-delle-stringhe

Capitolo 140: Serializzazione binaria

Osservazioni

Il motore di serializzazione binario fa parte del framework .NET, ma gli esempi qui riportati sono specifici di C #. Rispetto ad altri motori di serializzazione integrati nel framework .NET, il serializzatore binario è veloce ed efficiente e di solito richiede pochissimo codice aggiuntivo per farlo funzionare. Tuttavia, è anche meno tollerante alle modifiche al codice; cioè, se serializzi un oggetto e poi apporti una leggera modifica alla definizione dell'oggetto, probabilmente non lo deserializzi correttamente.

Examples

Rendere serializzabile un oggetto

Aggiungi l'attributo `[Serializable]` per contrassegnare un intero oggetto per la serializzazione binaria:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal DontSerializeThis;

    [OptionalField]
    public string Name;
}
```

Tutti i membri verranno serializzati a meno che non si `[NonSerialized]` esplicitamente utilizzando l'attributo `[NonSerialized]` . Nel nostro esempio, `X` , `Y` , `Z` e `Name` sono tutti serializzati.

Tutti i membri devono essere presenti alla deserializzazione a meno che non siano contrassegnati con `[NonSerialized]` o `[OptionalField] [NonSerialized] [OptionalField]` . Nel nostro esempio, `X` , `Y` e `Z` sono tutti richiesti e la deserializzazione fallirà se non sono presenti nello stream. `DontSerializeThis` sarà sempre impostato su `default(decimal)` (che è 0). Se `Name` è presente nello stream, verrà impostato su quel valore, altrimenti verrà impostato su `default(string)` (che è null). Lo scopo di `[OptionalField]` è di fornire un po 'di tolleranza della versione.

Controllo del comportamento di serializzazione con attributi

Se si utilizza l'attributo `[NonSerialized]` , quel membro avrà sempre il suo valore predefinito dopo la deserializzazione (ad esempio 0 per un `int` , null per `string` , false per un `bool` , ecc.), Indipendentemente dall'inizializzazione eseguita nell'oggetto stesso (costruttori, dichiarazioni,

ecc.). Per compensare, vengono `[OnDeserializing]` gli attributi `[OnDeserializing]` (chiamato solo BEFORE deserializing) e `[OnDeserialized]` (chiamato solo AFTER deserializing) insieme alle loro controparti, `[OnSerializing]` e `[OnSerialized]` .

Supponiamo di voler aggiungere un "Rating" al nostro Vector e vogliamo assicurarci che il valore inizi sempre a 1. Il modo in cui è scritto qui sotto, sarà 0 dopo essere stato deserializzato:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal Rating = 1M;

    public Vector()
    {
        Rating = 1M;
    }

    public Vector(decimal initialRating)
    {
        Rating = initialRating;
    }
}
```

Per risolvere questo problema, possiamo semplicemente aggiungere il seguente metodo all'interno della classe per impostarlo su 1:

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
    Rating = 1M;
}
```

Oppure, se vogliamo impostarlo su un valore calcolato, possiamo aspettare che finisca la deserializzazione e quindi impostarlo:

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
    Rating = 1 + ((X+Y+Z)/3);
}
```

Allo stesso modo, possiamo controllare come vengono scritte le cose usando `[OnSerializing]` e `[OnSerialized]` .

Aggiungere più controllo implementando ISerializable

Ciò otterrebbe un maggiore controllo sulla serializzazione, su come salvare e caricare i tipi

Implementare un'interfaccia ISerializable e creare un costruttore vuoto da compilare

```
[Serializable]
public class Item : ISerializable
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public Item ()
    {

    }

    protected Item (SerializationInfo info, StreamingContext context)
    {
        _name = (string)info.GetValue("_name", typeof(string));
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("_name", _name, typeof(string));
    }
}
```

Per la serializzazione dei dati, è possibile specificare il nome desiderato e il tipo desiderato

```
info.AddValue("_name", _name, typeof(string));
```

Quando i dati sono deserializzati, sarai in grado di leggere il tipo desiderato

```
_name = (string)info.GetValue("_name", typeof(string));
```

Surrogati di serializzazione (implementazione di ISerializationSurrogate)

Implementa un selettore surrogato di serializzazione che consente a un oggetto di eseguire la serializzazione e la deserializzazione di un altro

Inoltre consente di serializzare o deserializzare correttamente una classe che non è serializzabile

Implementare l'interfaccia ISerializationSurrogate

```
public class ItemSurrogate : ISerializationSurrogate
{
    public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
    {
        var item = (Item)obj;
        info.AddValue("_name", item.Name);
    }

    public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
    {
```

```
        var item = (Item)obj;
        item.Name = (string)info.GetValue("_name", typeof(string));
        return item;
    }
}
```

Quindi devi informare il tuo IFormatter dei surrogati definendo e inizializzando un SurrogateSelector e assegnandolo al tuo IFormatter

```
var surrogateSelector = new SurrogateSelector();
surrogateSelector.AddSurrogate(typeof(Item), new StreamingContext(StreamingContextStates.All),
new ItemSurrogate());
var binaryFormatter = new BinaryFormatter
{
    SurrogateSelector = surrogateSelector
};
```

Anche se la classe non è marcata serializzabile.

```
//this class is not serializable
public class Item
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

La soluzione completa

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class Item
    {
        private string _name;

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class ItemSurrogate : ISerializationSurrogate
    {
        public void GetObjectData(object obj, SerializationInfo info, StreamingContext
context)
        {
            var item = (Item)obj;
```

```
            info.AddValue("_name", item.Name);
        }

        public object SetObjectData(object obj, SerializationInfo info, StreamingContext
context, ISurrogateSelector selector)
        {
            var item = (Item)obj;
            item.Name = (string)info.GetValue("_name", typeof(string));
            return item;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var item = new Item
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

            var binaryFormatter = new BinaryFormatter
            {
                SurrogateSelector = surrogateSelector
            };

            using (var memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, obj);
                return memoryStream.ToArray();
            }
        }

        private static object DeserializeData(byte[] bytes)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

            var binaryFormatter = new BinaryFormatter
            {
                SurrogateSelector = surrogateSelector
            };

            using (var memoryStream = new MemoryStream(bytes))
                return binaryFormatter.Deserialize(memoryStream);
        }
    }
}
```

Serialization Binder

Il raccoglitore ti dà l'opportunità di verificare quali tipi vengono caricati nel dominio dell'applicazione

Creare una classe ereditata da SerializationBinder

```
class MyBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        if (typeName.Equals("BinarySerializationExample.Item"))
            return typeof(Item);
        return null;
    }
}
```

Ora possiamo controllare quali tipi stanno caricando e su questa base per decidere cosa vogliamo veramente ricevere

Per usare un raccoglitore, devi aggiungerlo a BinaryFormatter.

```
object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    binaryFormatter.Binder = new MyBinder();

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

La soluzione completa

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class MyBinder : SerializationBinder
    {
        public override Type BindToType(string assemblyName, string typeName)
        {
            if (typeName.Equals("BinarySerializationExample.Item"))
                return typeof(Item);
            return null;
        }
    }

    [Serializable]
    public class Item
    {
        private string _name;

        public string Name
```

```
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var item = new Item
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var binaryFormatter = new BinaryFormatter();
            using (var memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, obj);
                return memoryStream.ToArray();
            }
        }

        private static object DeserializeData(byte[] bytes)
        {
            var binaryFormatter = new BinaryFormatter
            {
                Binder = new MyBinder()
            };

            using (var memoryStream = new MemoryStream(bytes))
                return binaryFormatter.Deserialize(memoryStream);
        }
    }
}
```

Alcuni trucchi in retrocompatibilità

Questo piccolo esempio mostra come si può perdere la retrocompatibilità nei propri programmi se non si presta attenzione a ciò. E i modi per ottenere un maggiore controllo del processo di serializzazione

Inizialmente, scriveremo un esempio della prima versione del programma:

Versione 1

```
[Serializable]
class Data
{
    [OptionalField]
    private int _version;
```

```
    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }
}
```

E ora, supponiamo che nella seconda versione del programma sia stata aggiunta una nuova classe. E abbiamo bisogno di memorizzarlo in un array.

Ora il codice sarà simile a questo:

Versione 2

```
[Serializable]
class NewItem
{
    [OptionalField]
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }

    [OptionalField]
    private List<NewItem> _newItems;

    public List<NewItem> NewItems
    {
        get { return _newItems; }
        set { _newItems = value; }
    }
}
```

E codice per serializzare e deserializzare

```
private static byte[] SerializeData(object obj)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
```

```
    }
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

E quindi, cosa succederebbe quando serializzaste i dati nel programma di v2 e cerchereste di deserializzarli nel programma di v1?

Ottieni un'eccezione:

```
System.Runtime.Serialization.SerializationException was unhandled
Message=The ObjectManager found an invalid number of fixups. This usually indicates a problem
in the Formatter.Source=mscorlib
StackTrace:
   at System.Runtime.Serialization.ObjectManager.DoFixups()
   at System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler
handler, __BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
   at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
   at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream)
   at Microsoft.Samples.TestV1.Main(String[] args) in c:\Users\andrew\Documents\Visual Studio
2013\Projects\vts\CS\V1 Application\TestV1Part2\TestV1Part2.cs:line 29
   at System.AppDomain._nExecuteAssembly(Assembly assembly, String[] args)
   at Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback
callback, Object state)
   at System.Threading.ThreadHelper.ThreadStart()
```

Perché?

L'ObjectManager ha una logica diversa per risolvere le dipendenze per gli array e per i tipi di riferimento e valore. Abbiamo aggiunto una serie di nuovi il tipo di riferimento che è assente nel nostro assembly.

Quando ObjectManager tenta di risolvere le dipendenze costruisce il grafico. Quando vede la matrice, non può risolverlo immediatamente, in modo da creare un riferimento fittizio e quindi correggere l'array in un secondo momento.

E poiché questo tipo non è nell'assembly e le dipendenze non possono essere riparate. Per qualche ragione, non rimuove la matrice dall'elenco di elementi per le correzioni e alla fine genera un'eccezione "IncorrectNumberOfFixups".

Sono alcuni "trucchi" nel processo di serializzazione. Per qualche ragione, non funziona correttamente solo per le matrici di nuovi tipi di riferimento.

```
A Note:
```

```
Similar code will work correctly if you do not use arrays with new classes
```

E il primo modo per risolverlo e mantenere la compatibilità?

- Usa una collezione di nuove strutture piuttosto che classi o usa un dizionario (possibili classi), perché un dizionario è una raccolta di keyvaluepair (è la struttura)
- Usa ISerializable, se non puoi modificare il vecchio codice

Leggi Serializzazione binaria online: https://riptutorial.com/it/csharp/topic/4120/serializzazione-binaria

Capitolo 141: straripamento

Examples

Overflow intero

C'è una capacità massima che un numero intero può memorizzare. E quando supererai quel limite, tornerà al lato negativo. Per `int` , è `2147483647`

```
int x = int.MaxValue;                //MaxValue is 2147483647
x = unchecked(x + 1);                //make operation explicitly unchecked so that the example
also works when the check for arithmetic overflow/underflow is enabled in the project settings

Console.WriteLine(x);                //Will print -2147483648
Console.WriteLine(int.MinValue);     //Same as Min value
```

Per tutti gli interi al di fuori di questo intervallo utilizzare lo spazio dei nomi System.Numerics che ha il tipo di dati BigInteger. Controlla sotto il link per maggiori informazioni
https://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.110).aspx

Overflow durante il funzionamento

L'overflow si verifica anche durante l'operazione. Nell'esempio seguente, x è un `int` , 1 è un `int` di default. Quindi l'aggiunta è un'aggiunta `int` . E il risultato sarà un `int` . E traboccherà.

```
int x = int.MaxValue;               //MaxValue is 2147483647
long y = x + 1;                     //It will be overflown
Console.WriteLine(y);               //Will print -2147483648
Console.WriteLine(int.MinValue);    //Same as Min value
```

Puoi evitarlo usando 1L. Ora 1 sarà `long` e l'aggiunta sarà una `long` aggiunta

```
int x = int.MaxValue;               //MaxValue is 2147483647
long y = x + 1L;                    //It will be OK
Console.WriteLine(y);               //Will print 2147483648
```

Ordinare è importante

C'è overflow nel seguente codice

```
int x = int.MaxValue;
Console.WriteLine(x + x + 1L);  //prints -1
```

Mentre nel codice seguente non c'è overflow

```
int x = int.MaxValue;
Console.WriteLine(x + 1L + x);  //prints 4294967295
```

Ciò è dovuto all'ordinamento da sinistra a destra delle operazioni. Nel primo frammento di codice `x + x` trabocca e dopo diventa `long` . D'altra parte `x + 1L` diventa `long` e dopo che `x` viene aggiunto a questo valore.

Leggi straripamento online: https://riptutorial.com/it/csharp/topic/3303/straripamento

Capitolo 142: String Concatenate

Osservazioni

Se stai creando una stringa dinamica, è buona norma optare per la classe `StringBuilder` piuttosto che unire stringhe usando il metodo + o `Concat` poiché ogni + / `Concat` crea un nuovo oggetto stringa ogni volta che viene eseguito.

Examples

+ Operatore

```
string s1 = "string1";
string s2 = "string2";

string s3 = s1 + s2; // "string1string2"
```

Concatena le stringhe usando System.Text.StringBuilder

La concatenazione di stringhe utilizzando uno StringBuilder può offrire vantaggi in termini di prestazioni rispetto alla semplice concatenazione di stringhe usando `+` . Ciò è dovuto al modo in cui la memoria è allocata. Le stringhe sono riallocate con ogni concatenazione, StringBuilders alloca la memoria nei blocchi solo riallocando quando il blocco corrente è esaurito. Questo può fare una grande differenza quando si fanno molte piccole concatenazioni.

```
StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++)
{
    sb.Append(i);
    sb.Append(" ");
}
Console.WriteLine(sb.ToString()); // "1 2 3 4 5 "
```

Le chiamate a `Append()` possono essere collegate in cascata, in quanto restituisce un riferimento a `StringBuilder` :

```
StringBuilder sb = new StringBuilder();
sb.Append("some string ")
  .Append("another string");
```

Elementi dell'array stringa Concat che utilizzano String.Join

Il metodo `String.Join` può essere utilizzato per concatenare più elementi da una matrice di stringhe.

```
string[] value = {"apple", "orange", "grape", "pear"};
```

```
string separator = ", ";

string result = String.Join(separator, value, 1, 2);
Console.WriteLine(result);
```

Produce il seguente output: "orange, grape"

Questo esempio utilizza il `String.Join(String, String[], Int32, Int32)` , che specifica l'indice iniziale e il conteggio in cima al separatore e al valore.

Se non si desidera utilizzare startIndex e contare sovraccarichi, è possibile unire tutte le stringhe fornite. Come questo:

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
string result = String.Join(separator, value);
Console.WriteLine(result);
```

che produrrà;

mela, arancia, uva, pera

Concatenazione di due stringhe usando $

$ fornisce un metodo semplice e conciso per concatenare più stringhe.

```
var str1 = "text1";
var str2 = " ";
var str3 = "text3";
string result2 = $"{str1}{str2}{str3}"; //"text1 text3"
```

Leggi String Concatenate online: https://riptutorial.com/it/csharp/topic/3616/string-concatenate

Capitolo 143: String.Format

introduzione

I metodi `Format` sono una serie di overload nella classe `System.String` utilizzata per creare stringhe che combinano oggetti in rappresentazioni di stringa specifiche. Queste informazioni possono essere applicate a `String.Format` , a vari metodi `WriteLine` e ad altri metodi nel framework .NET.

Sintassi

- string.Format (formato stringa, oggetto params [] args)
- string.Format (provider IFormatProvider, formato stringa, oggetto params [] args)
- $ "stringa {testo} blablabla" // Poiché C # 6

Parametri

Parametro	Dettagli
formato	Una stringa di formato composita , che definisce il modo in cui gli *argomenti* devono essere combinati in una stringa.
args	Una sequenza di oggetti da combinare in una stringa. Poiché utilizza un argomento `params` , puoi utilizzare un elenco di argomenti separato da virgola o un array di oggetti effettivo.
fornitore	Una raccolta di modi per formattare gli oggetti alle stringhe. I valori tipici includono CultureInfo.InvariantCulture and CultureInfo.CurrentCulture .

Osservazioni

Gli appunti:

- `String.Format()` gestisce gli argomenti `null` senza `String.Format()` un'eccezione.
- Esistono sovraccarichi che sostituiscono il parametro `args` con uno, due o tre parametri dell'oggetto.

Examples

Luoghi in cui String.Format è "incorporato" nel framework

Esistono diversi punti in cui è possibile utilizzare `String.Format` *indirettamente* : il segreto è cercare il sovraccarico con il `string format, params object[] args` firma `string format, params object[] args` , ad esempio:

```
Console.WriteLine(String.Format("{0} - {1}", name, value));
```

Può essere sostituito con una versione più corta:

```
Console.WriteLine("{0} - {1}", name, value);
```

Esistono altri metodi che usano anche `String.Format` ad esempio:

```
Debug.WriteLine(); // and Print()
StringBuilder.AppendFormat();
```

Utilizzando il formato numerico personalizzato

`NumberFormatInfo` può essere utilizzato per formattare sia numeri interi che numeri a virgola mobile.

```
// invariantResult is "1,234,567.89"
var invarianResult = string.Format(CultureInfo.InvariantCulture, "{0:#,###,##}", 1234567.89);

// NumberFormatInfo is one of classes that implement IFormatProvider
var customProvider = new NumberFormatInfo
{
    NumberDecimalSeparator = "_NS_", // will be used instead of ','
    NumberGroupSeparator = "_GS_", // will be used instead of '.'
};

// customResult is "1_GS_234_GS_567_NS_89"
var customResult = string.Format(customProvider, "{0:#,###.##}", 1234567.89);
```

Creare un fornitore di formato personalizzato

```
public class CustomFormat : IFormatProvider, ICustomFormatter
{
    public string Format(string format, object arg, IFormatProvider formatProvider)
    {
        if (!this.Equals(formatProvider))
        {
            return null;
        }

        if (format == "Reverse")
        {
            return String.Join("", arg.ToString().Reverse());
        }

        return arg.ToString();
    }

    public object GetFormat(Type formatType)
    {
        return formatType==typeof(ICustomFormatter) ? this:null;
    }
}
```

Uso:

```
String.Format(new CustomFormat(), "-> {0:Reverse} <-", "Hello World");
```

Produzione:

```
-> dlroW olleH <-
```

Allinea a sinistra / a destra, pad con spazi

Il secondo valore nelle parentesi graffe determina la lunghezza della stringa di sostituzione. Regolando il secondo valore come positivo o negativo, è possibile modificare l'allineamento della stringa.

```
string.Format("LEFT:  string: ->{0,-5}<- int: ->{1,-5}<-", "abc", 123);
string.Format("RIGHT: string: ->{0,5}<- int: ->{1,5}<-", "abc", 123);
```

Produzione:

```
LEFT:  string: ->abc  <- int: ->123  <-
RIGHT: string: ->  abc<- int: ->  123<-
```

Formati numerici

```
// Integral types as hex
string.Format("Hexadecimal: byte2: {0:x2}; byte4: {0:X4}; char: {1:x2}", 123, (int)'A');

// Integers with thousand separators
string.Format("Integer, thousand sep.: {0:#,#}; fixed length: >{0,10:#,#}<", 1234567);

// Integer with leading zeroes
string.Format("Integer, leading zeroes: {0:00}; ", 1);

// Decimals
string.Format("Decimal, fixed precision: {0:0.000}; as percents: {0:0.00%}", 0.12);
```

Produzione:

```
Hexadecimal: byte2: 7b; byte4: 007B; char: 41
Integer, thousand sep.: 1,234,567; fixed length: > 1,234,567<
Integer, leading zeroes: 01;
Decimal, fixed precision: 0.120; as percents: 12.00%
```

Formattazione della valuta

L'identificatore di formato "c" (o valuta) converte un numero in una stringa che rappresenta un importo in valuta.

```
string.Format("{0:c}", 112.236677) // $112.23 - defaults to system
```

Precisione

Il valore predefinito è 2. Usa c1, c2, c3 e così via per controllare la precisione.

```
string.Format("{0:C1}", 112.236677) //$112.2
string.Format("{0:C3}", 112.236677) //$112.237
string.Format("{0:C4}", 112.236677) //$112.2367
string.Format("{0:C9}", 112.236677) //$112.236677000
```

Simbolo di valuta

1. Passa l'istanza di `CultureInfo` per usare il simbolo di cultura personalizzato.

```
string.Format(new CultureInfo("en-US"), "{0:c}", 112.236677); //$112.24
string.Format(new CultureInfo("de-DE"), "{0:c}", 112.236677); //112,24 €
string.Format(new CultureInfo("hi-IN"), "{0:c}", 112.236677); //□ 112.24
```

2. Usa qualsiasi stringa come simbolo di valuta. Usa `NumberFormatInfo` per personalizzare il simbolo di valuta.

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi = (NumberFormatInfo) nfi.Clone();
nfi.CurrencySymbol = "?";
string.Format(nfi, "{0:C}", 112.236677); //?112.24
nfi.CurrencySymbol = "?%^&";
string.Format(nfi, "{0:C}", 112.236677); //?%^&112.24
```

Posizione del simbolo di valuta

Utilizzare CurrencyPositivePattern per i valori positivi e CurrencyNegativePattern per i valori negativi.

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi.CurrencyPositivePattern = 0;
string.Format(nfi, "{0:C}", 112.236677); //$112.24 - default
nfi.CurrencyPositivePattern = 1;
string.Format(nfi, "{0:C}", 112.236677); //112.24$
nfi.CurrencyPositivePattern = 2;
string.Format(nfi, "{0:C}", 112.236677); //$ 112.24
nfi.CurrencyPositivePattern = 3;
string.Format(nfi, "{0:C}", 112.236677); //112.24 $
```

L'utilizzo del pattern negativo è lo stesso del pattern positivo. Molto più casi d'uso si prega di fare riferimento al link originale.

Separatore decimale personalizzato

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
```

```
nfi.CurrencyPositivePattern = 0;
nfi.CurrencyDecimalSeparator = "..";
string.Format(nfi, "{0:C}", 112.236677); //$112..24
```

Dal momento che C # 6.0

6.0

Dal momento che C # 6.0 è possibile utilizzare l'interpolazione delle stringhe al posto di `String.Format` .

```
string name = "John";
string lastname = "Doe";
Console.WriteLine($"Hello {name} {lastname}!");
```

Ciao John Doe!

Altri esempi di questo argomento sono disponibili nell'argomento C # 6.0: Interpolazione di stringhe .

Sfuggire parentesi graffe all'interno di un'espressione String.Format ()

```
string outsidetext = "I am outside of bracket";
string.Format("{{I am in brackets!}} {0}", outsidetext);

//Outputs "{I am in brackets!} I am outside of bracket"
```

Formattazione della data

```
DateTime date = new DateTime(2016, 07, 06, 18, 30, 14);
// Format: year, month, day hours, minutes, seconds

Console.Write(String.Format("{0:dd}",date));

//Format by Culture info
String.Format(new System.Globalization.CultureInfo("mn-MN"),"{0:dddd}",date);
```

6.0

```
Console.Write($"{date:ddd}");
```

produzione :

```
06
Лхагва
06
```

specifier	Senso	Campione	Risultato
d	Data	`{0:d}`	2016/07/06
dd	Giorno, zero imbottito	`{0:dd}`	06

specifier	Senso	Campione	Risultato
ddd	Nome del giorno breve	`{0:ddd}`	sposare
dddd	Nome per l'intera giornata	`{0:dddd}`	mercoledì
D	Un appuntamento lungo	`{0:D}`	Mercoledì 6 luglio 2016
f	Data e ora completi, a breve	`{0:f}`	Mercoledì 6 luglio 2016 18.30
ff	Seconda frazione, 2 cifre	`{0:ff}`	20
F F F	Seconda frazione, 3 cifre	`{0:fff}`	201
ffff	Seconda frazione, 4 cifre	`{0:ffff}`	2016
F	Data e ora complete, lunghe	`{0:F}`	Mercoledì 6 luglio 2016 6:30:14 PM
g	Data e ora predefinite	`{0:g}`	7/6/2016 6:30 PM
gg	Era	`{0:gg}`	ANNO DOMINI
hh	Ora (2 cifre, 12 ore)	`{0:hh}`	06
HH	Ora (2 cifre, 24 ore)	`{0:HH}`	18
M	Mese e giorno	`{0:M}`	6 luglio
mm	Minuti, senza riempimento	`{0:mm}`	30
MM	Mese, riempito a zero	`{0:MM}`	07
MMM	Nome del mese di 3 lettere	`{0:MMM}`	luglio
MMMM	Nome completo del mese	`{0:MMMM}`	luglio
ss	secondi	`{0:ss}`	14
r	Data RFC1123	`{0:r}`	Mer, 06 lug 2016 18:30:14 GMT
S	Stringa di data ordinabile	`{0:s}`	2016-07-06T18: 30: 14
t	Poco tempo	`{0:t}`	6:30 PM
T	A lungo	`{0:T}`	6:30:14 PM
TT	AM PM	`{0:tt}`	PM
u	Ora locale ordinabile universale	`{0:u}`	18-07-2016 18: 30: 14Z

specifier	Senso	Campione	Risultato
U	GMT universale	`{0:U}`	Mercoledì 6 luglio 2016 9:30:14
Y	Mese e anno	`{0:Y}`	Luglio 2016
aa	Anno 2 cifre	`{0:yy}`	16
aaaa	Anno 4 cifre	`{0:yyyy}`	2016
zz	Offset fuso orario a 2 cifre	`{0:zz}`	+09
zzz	offset fuso orario completo	`{0:zzz}`	+09: 00

Accordare()

Il metodo ToString () è presente su tutti i tipi di oggetti di riferimento. Ciò è dovuto al fatto che tutti i tipi di riferimento derivano da Object su cui è presente il metodo ToString (). Il metodo ToString () sulla classe base dell'oggetto restituisce il nome del tipo. Il frammento seguente stamperà "Utente" sulla console.

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
}

...

var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

Tuttavia, la classe User può anche sovrascrivere ToString () per modificare la stringa restituita. Il frammento di codice sottostante stampa "Id: 5, Name: User1" sulla console.

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
    public override ToString()
    {
        return string.Format("Id: {0}, Name: {1}", Id, Name);
    }
}

...

var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

Relazione con ToString ()

Mentre il metodo `String.Format()` è certamente utile per la formattazione dei dati come stringhe,

può spesso essere un po 'eccessivo, specialmente quando si ha a che fare con un singolo oggetto come mostrato di seguito:

```
String.Format("{0:C}", money);  // yields "$42.00"
```

Un approccio più semplice potrebbe essere quello di utilizzare semplicemente il metodo `ToString()` disponibile su tutti gli oggetti all'interno di C #. Supporta tutte le stesse stringhe di formattazione standard e personalizzate , ma non richiede la mappatura dei parametri necessaria in quanto ci sarà un solo argomento:

```
money.ToString("C");  // yields "$42.00"
```

Avvertenze e restrizioni alla formattazione

Sebbene questo approccio possa essere più semplice in alcuni scenari, l'approccio `ToString()` è limitato per quanto riguarda l'aggiunta del `String.Format()` sinistro o destro come si farebbe nel metodo `String.Format()` :

```
String.Format("{0,10:C}", money);  // yields "    $42.00"
```

Per ottenere lo stesso comportamento con il metodo `ToString()` , è necessario utilizzare un altro metodo come `PadLeft()` o `PadRight()` rispettivamente:

```
money.ToString("C").PadLeft(10);  // yields "    $42.00"
```

Leggi String.Format online: https://riptutorial.com/it/csharp/topic/79/string-format

Capitolo 144: StringBuilder

Examples

Che cos'è un oggetto StringBuilder e quando usarne uno

Un `StringBuilder` rappresenta una serie di caratteri, che a differenza di una stringa normale, sono mutabili. Spesso è necessario modificare le stringhe che abbiamo già creato, ma l'oggetto stringa standard non è mutabile. Ciò significa che ogni volta che una stringa viene modificata, è necessario creare un nuovo oggetto stringa, copiarlo e quindi riassegnarlo.

```
string myString = "Apples";
mystring += " are my favorite fruit";
```

Nell'esempio precedente, `myString` inizialmente ha solo il valore `"Apples"` . Tuttavia, quando concateniamo "è il mio frutto preferito", ciò che la classe di stringhe deve fare internamente implica:

- Creare una nuova matrice di caratteri uguale alla lunghezza di `myString` e alla nuova stringa che stiamo aggiungendo.
- Copia tutti i caratteri di `myString` all'inizio del nostro nuovo array e copia la nuova stringa nella fine dell'array.
- Crea un nuovo oggetto stringa in memoria e riassegnalo a `myString` .

Per una singola concatenazione, questo è relativamente banale. Tuttavia, cosa succederebbe se fosse necessario eseguire molte operazioni di accodamento, ad esempio in un ciclo?

```
String myString = "";
for (int i = 0; i < 10000; i++)
    myString += " "; // puts 10,000 spaces into our string
```

A causa della copia ripetuta e della creazione di oggetti, ciò determinerà un degrado significativo delle prestazioni del nostro programma. Possiamo evitarlo usando invece un `StringBuilder` .

```
StringBuilder myStringBuilder = new StringBuilder();
for (int i = 0; i < 10000; i++)
    myStringBuilder.Append(' ');
```

Ora, quando viene eseguito lo stesso ciclo, le prestazioni e la velocità del tempo di esecuzione del programma saranno significativamente più veloci rispetto all'utilizzo di una stringa normale. Per ripristinare `StringBuilder` in una stringa normale, possiamo semplicemente chiamare il metodo `ToString()` di `StringBuilder` .

Tuttavia, questa non è l'unica ottimizzazione che `StringBuilder` ha. Per ottimizzare ulteriormente le funzioni, possiamo sfruttare altre proprietà che aiutano a migliorare le prestazioni.

```
StringBuilder sb = new StringBuilder(10000); // initializes the capacity to 10000
```

Se sappiamo in anticipo quanto deve essere il nostro `StringBuilder` , possiamo specificare la sua dimensione in anticipo, il che impedirà di ridimensionare l'array di caratteri che ha internamente.

```
sb.Append('k', 2000);
```

Sebbene l'uso di `StringBuilder` per l'aggiunta sia molto più veloce di una stringa, può essere eseguito anche più velocemente se è necessario aggiungere un singolo carattere molte volte.

Una volta completata la creazione della stringa, è possibile utilizzare il metodo `ToString()` su `StringBuilder` per convertirlo in una `string` base. Questo è spesso necessario perché la classe `StringBuilder` non eredita dalla `string` .

Ad esempio, ecco come utilizzare `StringBuilder` per creare una `string` :

```
string RepeatCharacterTimes(char character, int times)
{
    StringBuilder builder = new StringBuilder("");
    for (int counter = 0; counter < times; counter++)
    {
        //Append one instance of the character to the StringBuilder.
        builder.Append(character);
    }
    //Convert the result to string and return it.
    return builder.ToString();
}
```

In conclusione, `StringBuilder` dovrebbe essere usato al posto della stringa quando è necessario apportare molte modifiche a una stringa con le prestazioni in mente.

Utilizzare StringBuilder per creare una stringa da un numero elevato di record

```
public string GetCustomerNamesCsv()
{
    List<CustomerData> customerDataRecords = GetCustomerData(); // Returns a large number of
records, say, 10000+

    StringBuilder customerNamesCsv = new StringBuilder();
    foreach (CustomerData record in customerDataRecords)
    {
       customerNamesCsv
            .Append(record.LastName)
            .Append(',')
            .Append(record.FirstName)
            .Append(Environment.NewLine);
    }

    return customerNamesCsv.ToString();
}
```

Leggi StringBuilder online: https://riptutorial.com/it/csharp/topic/4675/stringbuilder

Capitolo 145: Stringhe Verbatim

Sintassi

- @ "le stringhe verbatim sono stringhe il cui contenuto non è sfuggito, quindi in questo caso \ n non rappresenta il carattere di nuova riga ma due caratteri individuali: \ e n Le stringhe di Verbatim vengono create precedendo il contenuto della stringa con il carattere @"
- @ "Per sfuggire alle virgolette", vengono utilizzate le "virgolette doppie" ".

Osservazioni

Per concatenare i valori letterali stringa, utilizzare il simbolo @ all'inizio di ogni stringa.

```
var combinedString = @"\t means a tab" + @" and \n means a newline";
```

Examples

Stringhe multilinea

```
var multiLine = @"This is a

multiline paragraph";
```

Produzione:

> Questo è un
>
> paragrafo multilinea

Live Demo su .NET Fiddle

Anche le stringhe multilinea che contengono virgolette possono essere sfuggite come se fossero su una singola riga, perché sono stringhe letterali.

```
var multilineWithDoubleQuotes = @"I went to a city named

                        ""San Diego""

                    during summer vacation.";
```

Live Demo su .NET Fiddle

Si noti che gli spazi / tabulazioni all'inizio delle righe 2 e 3 qui sono effettivamente presenti nel valore della variabile; controlla questa domanda per possibili soluzioni.

Escaping Double Quotes

Le doppie virgolette all'interno di stringhe letterali possono essere sfuggite utilizzando 2 virgolette doppie `""` per rappresentare una virgoletta doppia `"` nella stringa risultante.

```
var str = @"""I don't think so,"" he said.";
Console.WriteLine(str);
```

Produzione:

> "Non penso", ha detto.

Live Demo su .NET Fiddle

Stringhe Verbatim interpolate

Le stringhe Verbatim possono essere combinate con le nuove funzionalità di interpolazione String trovate in C # 6.

```
Console.WriteLine($@"Testing \n 1 2 {5 - 2}
New line");
```

Produzione:

> Test \ n 1 2 3
> Nuova linea

Live Demo su .NET Fiddle

Come previsto da una stringa testuale, i backslash vengono ignorati come caratteri di escape. E come previsto da una stringa interpolata, qualsiasi espressione all'interno di parentesi graffe viene valutata prima di essere inserita nella stringa in quella posizione.

Le stringhe Verbatim istruiscono il compilatore a non utilizzare i caratteri di escape

In una stringa normale, il carattere backslash è il carattere di escape, che indica al compilatore di guardare il / i carattere / i successivo / i per determinare il carattere effettivo nella stringa. (Elenco completo di caratteri escape)

Nelle stringhe letterali, non ci sono caratteri di escape (eccetto per `""` che è trasformato in un `"`). Per usare una stringa testuale, basta anteporre un `@` prima delle virgolette iniziali.

Questa stringa letterale

```
var filename = @"c:\temp\newfile.txt"
```

Produzione:

c: \ temp \ newfile.txt

Al contrario di usare una stringa ordinaria (non verbale):

```
var filename = "c:\temp\newfile.txt"
```

che produrrà:

```
c:    emp
ewfile.txt
```

usando caratteri che escaping. (Il `\t` viene sostituito con un carattere di tabulazione e `\n` viene sostituito con un carattere di fine riga.)

Live Demo su .NET Fiddle

Leggi Stringhe Verbatim online: https://riptutorial.com/it/csharp/topic/16/stringhe-verbatim

Capitolo 146: Structs

Osservazioni

A differenza delle classi, una `struct` è un tipo di valore, e viene creata sullo stack locale e non sull'heap gestito, *per impostazione predefinita* . Ciò significa che una volta che lo stack specifico esce dall'ambito, la `struct` viene `struct` . Anche i tipi di riferimento `struct` nenti alle `struct` vengono spazzati, una volta che il GC determina che non sono più riferiti dalla `struct` .

`struct` s non può ereditare e non può essere base per ereditarietà, sono implicitamente sigillate e non possono includere membri `protected` . Tuttavia, una `struct` può implementare un'interfaccia, come fanno le classi.

Examples

Dichiarazione di una struttura

```
public struct Vector
{
    public int X;
    public int Y;
    public int Z;
}

public struct Point
{
    public decimal x, y;

    public Point(decimal pointX, decimal pointY)
    {
        x = pointX;
        y = pointY;
    }
}
```

- `struct` campi istanza `struct` possono essere impostati tramite un costruttore parametrizzato o singolarmente dopo la costruzione di `struct` .
- I membri privati possono essere inizializzati solo dal costruttore.
- `struct` definisce un tipo sealed che eredita implicitamente da System.ValueType.
- Le strutture non possono ereditare da nessun altro tipo, ma possono implementare interfacce.
- Le strutture vengono copiate sull'assegnazione, ovvero tutti i dati vengono copiati nella nuova istanza e le modifiche a una di esse non vengono riflesse dall'altra.
- Una struct non può essere `null` , sebbene *possa essere* usato come un tipo nullable:

```
Vector v1 = null; //illegal
Vector? v2 = null; //OK
Nullable<Vector> v3 = null // OK
```

- Le strutture possono essere istanziate con o senza l'utilizzo del `new` operatore.

```
//Both of these are acceptable
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Vector v2;
v2.X = 1;
v2.Y = 2;
v2.Z = 3;
```

  Tuttavia, è necessario utilizzare il `new` operatore per utilizzare un inizializzatore:

```
Vector v1 = new MyStruct { X=1, Y=2, Z=3 }; // OK
Vector v2 { X=1, Y=2, Z=3 }; // illegal
```

Una struct può dichiarare tutto ciò che una classe può dichiarare, con alcune eccezioni:

- Una struct non può dichiarare un costruttore senza parametri. `struct` campi istanza `struct` possono essere impostati tramite un costruttore parametrizzato o singolarmente dopo la costruzione di `struct` . I membri privati possono essere inizializzati solo dal costruttore.
- Una struct non può dichiarare i membri come protetti, dal momento che è implicitamente sigillato.
- I campi Struct possono essere inizializzati solo se costanti o statici.

Struct utilizzo

Con il costruttore:

```
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=1,Y=2,Z=3

Vector v1 = new Vector();
//v1.X is not assigned
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=0,Y=2,Z=3

Point point1 = new Point();
point1.x = 0.5;
point1.y = 0.6;
```

```
Point point2 = new Point(0.5, 0.6);
```

Senza costruttore:

```
Vector v1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
//Output ERROR "Use of possibly unassigned field 'X'

Vector v1;
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=1,Y=2,Z=3

Point point3;
point3.x = 0.5;
point3.y = 0.6;
```

Se usiamo una struct con il suo costruttore, non avremo problemi con il campo non assegnato (ogni campo non assegnato ha valore null).

A differenza delle classi, non è necessario costruire una struttura, ovvero non è necessario utilizzare la nuova parola chiave, a meno che non sia necessario chiamare uno dei costruttori. Una struct non richiede la nuova parola chiave perché è un tipo di valore e quindi non può essere nullo.

Struct interfaccia di implementazione

```
public interface IShape
{
    decimal Area();
}

public struct Rectangle : IShape
{
    public decimal Length { get; set; }
    public decimal Width { get; set; }

    public decimal Area()
    {
        return Length * Width;
    }
}
```

Le strutture sono copiate sul compito

Le strutture sinse sono tipi di valore, tutti i dati vengono *copiati* durante l'assegnazione e qualsiasi modifica alla nuova copia non modifica i dati per la copia originale. Il frammento di codice

seguente mostra che `p1` è *copiato* in `p2` e le modifiche apportate su `p1` non influenzano l'istanza di `p2` .

```
var p1 = new Point {
    x = 1,
    y = 2
};

Console.WriteLine($"{p1.x} {p1.y}"); // 1 2

var p2 = p1;
Console.WriteLine($"{p2.x} {p2.y}"); // Same output: 1 2

p1.x = 3;
Console.WriteLine($"{p1.x} {p1.y}"); // 3 2
Console.WriteLine($"{p2.x} {p2.y}"); // p2 remain the same: 1 2
```

Leggi Structs online: https://riptutorial.com/it/csharp/topic/778/structs

Capitolo 147: System.DirectoryServices.Protocols.LdapConnec

Examples

Connessione LDAP SSL autenticata, il certificato SSL non corrisponde al DNS inverso

Imposta alcune costanti per il server e le informazioni di autenticazione. Supponendo LDAPv3, ma è abbastanza facile cambiarlo.

```
// Authentication, and the name of the server.
private const string LDAPUser =
"cn=example:app:mygroup:accts,ou=Applications,dc=example,dc=com";
private readonly char[] password = { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd' };
private const string TargetServer = "ldap.example.com";

// Specific to your company. Might start "cn=manager" instead of "ou=people", for example.
private const string CompanyDN = "ou=people,dc=example,dc=com";
```

Effettivamente creare la connessione con tre parti: un LdapDirectoryIdentifier (il server) e NetworkCredentials.

```
// Configure server and port. LDAP w/ SSL, aka LDAPS, uses port 636.
// If you don't have SSL, don't give it the SSL port.
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer, 636);

// Configure network credentials (userid and password)
var secureString = new SecureString();
foreach (var character in password)
        secureString.AppendChar(character);
NetworkCredential creds = new NetworkCredential(LDAPUser, secureString);

// Actually create the connection
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType = AuthType.Basic,
    SessionOptions =
    {
        ProtocolVersion = 3,
        SecureSocketLayer = true
    }
};

// Override SChannel reverse DNS lookup.
// This gets us past the "The LDAP server is unavailable." exception
// Could be
//    connection.SessionOptions.VerifyServerCertificate += { return true; };
// but some certificate validation is probably good.
connection.SessionOptions.VerifyServerCertificate +=
    (sender, certificate) => certificate.Subject.Contains(string.Format("CN={0},",
TargetServer));
```

Utilizzare il server LDAP, ad es. Cercare qualcuno tramite userid per tutti i valori objectClass. L'oggetto objectClass è presente per dimostrare una ricerca composta: la e commerciale è l'operatore booleano "e" per le due clausole di query.

```
 SearchRequest searchRequest = new SearchRequest(
        CompanyDN,
        string.Format((&(objectClass=*)(uid={0}))), uid),
        SearchScope.Subtree,
        null
);

// Look at your results
foreach (SearchResultEntry entry in searchResponse.Entries) {
    // do something
}
```

Super semplice anonimo LDAP

Supponendo LDAPv3, ma è abbastanza facile cambiarlo. Questa è una creazione LDAPv3 LdapConnection anonima, non crittografata.

```
private const string TargetServer = "ldap.example.com";
```

Effettivamente creare la connessione con tre parti: un LdapDirectoryIdentifier (il server) e NetworkCredentials.

```
// Configure server and credentials
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer);
NetworkCredential creds = new NetworkCredential();
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType=AuthType.Anonymous,
    SessionOptions =
    {
        ProtocolVersion = 3
    }
};
```

Per usare la connessione, qualcosa del genere avrebbe portato le persone al cognome Smith

```
SearchRequest searchRequest = new SearchRequest("dn=example,dn=com", "(sn=Smith)",
SearchScope.Subtree,null);
```

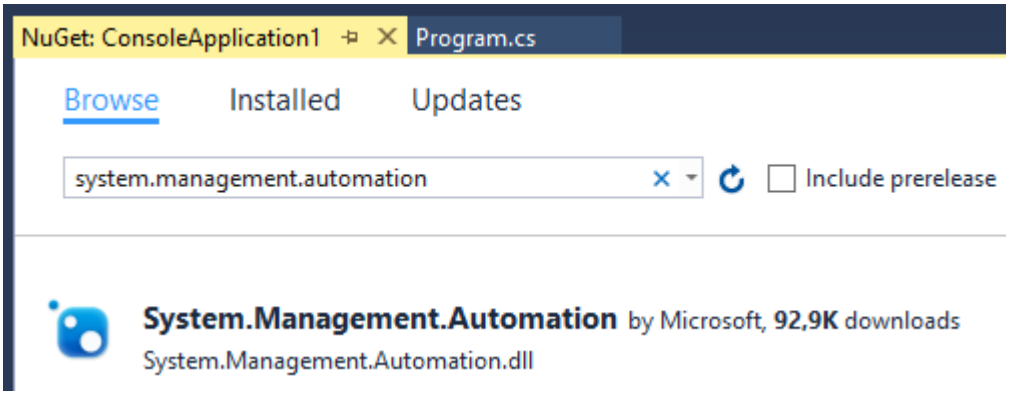
Leggi System.DirectoryServices.Protocols.LdapConnection online:
https://riptutorial.com/it/csharp/topic/5177/system-directoryservices-protocols-ldapconnection

Capitolo 148: System.Management.Automation

Osservazioni

> Lo spazio dei nomi *System.Management.Automation* è lo spazio dei nomi di root per Windows PowerShell.

System.Management.Automation è una libreria di estensione di Microsoft e può essere aggiunta ai progetti di Visual Studio tramite il gestore pacchetti NuGet o la console di gestione pacchetti.



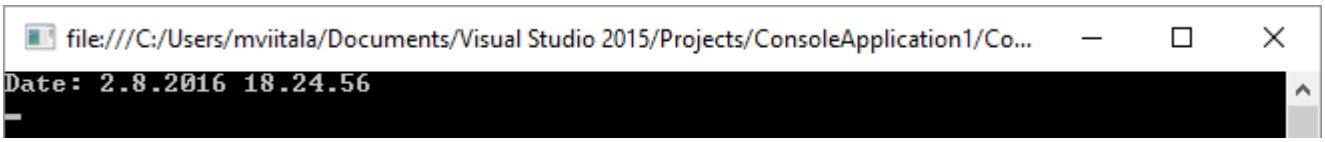
```
PM> Install-Package System.Management.Automation
```

Examples

Richiama la semplice pipeline sincrona

Ottieni la data e l'ora correnti.

```
public class Program
{
    static void Main()
    {
        // create empty pipeline
        PowerShell ps = PowerShell.Create();

        // add command
        ps.AddCommand("Get-Date");

        // run command(s)
        Console.WriteLine("Date: {0}", ps.Invoke().First());

        Console.ReadLine();
    }
}
```



Leggi System.Management.Automation online: https://riptutorial.com/it/csharp/topic/4988/system-management-automation

Capitolo 149: Task Libreria parallela

Examples

Parallel.ForEach

Un esempio che utilizza il ciclo Parallel.ForEach per eseguire il ping di un determinato array di URL del sito Web.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.ForEach(urls, url =>
    {
        var ping = new System.Net.NetworkInformation.Ping();

        var result = ping.Send(url);

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", url));
        }
    });
}
```

Parallel.For

Un esempio che utilizza Parallel.Per il ciclo per eseguire il ping di una determinata matrice di URL del sito Web.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.For(0, urls.Length, i =>
    {
        var ping = new System.Net.NetworkInformation.Ping();

        var result = ping.Send(urls[i]);
```

```
        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", urls[i]));
        }
    });
}
```

Parallel.Invoke

Richiamo di metodi o azioni in parallelo (regione parallela)

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.Invoke(
        () => PingUrl(urls[0]),
        () => PingUrl(urls[1]),
        () => PingUrl(urls[2]),
        () => PingUrl(urls[3])
    );
}

void PingUrl(string url)
{
    var ping = new System.Net.NetworkInformation.Ping();

    var result = ping.Send(url);

    if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
    {
        Console.WriteLine(string.Format("{0} is online", url));
    }
}
```

Un compito di polling cancellabile asincrono che attende tra le iterazioni

```
public class Foo
{
    private const int TASK_ITERATION_DELAY_MS = 1000;
    private CancellationTokenSource _cts;

    public Foo()
    {
        this._cts = new CancellationTokenSource();
    }

    public void StartExecution()
    {
        Task.Factory.StartNew(this.OwnCodeCancelableTask_EveryNSeconds, this._cts.Token);
    }
```

```
    public void CancelExecution()
    {
        this._cts.Cancel();
    }

    /// <summary>
    /// "Infinite" loop that runs every N seconds. Good for checking for a heartbeat or
updates.
    /// </summary>
    /// <param name="taskState">The cancellation token from our _cts field, passed in the
StartNew call</param>
    private async void OwnCodeCancelableTask_EveryNSeconds(object taskState)
    {
        var token = (CancellationToken)taskState;

        while (!token.IsCancellationRequested)
        {
            Console.WriteLine("Do the work that needs to happen every N seconds in this
loop");

            // Passing token here allows the Delay to be cancelled if your task gets
cancelled.
            await Task.Delay(TASK_ITERATION_DELAY_MS, token);
        }
    }
}
```

Un compito di polling cancellabile usando CancellationTokenSource

```
public class Foo
{
    private CancellationTokenSource _cts;

    public Foo()
    {
        this._cts = new CancellationTokenSource();
    }

    public void StartExecution()
    {
        Task.Factory.StartNew(this.OwnCodeCancelableTask, this._cts.Token);
    }

    public void CancelExecution()
    {
        this._cts.Cancel();
    }

    /// <summary>
    /// "Infinite" loop with no delays. Writing to a database while pulling from a buffer for
example.
    /// </summary>
    /// <param name="taskState">The cancellation token from our _cts field, passed in the
StartNew call</param>
    private void OwnCodeCancelableTask(object taskState)
    {
        var token = (CancellationToken) taskState; //Our cancellation token passed from
StartNew();
```

```
        while ( !token.IsCancellationRequested )
        {
            Console.WriteLine("Do your task work in this loop");
        }
    }
}
```

Versione asincrona di PingUrl

```
    static void Main(string[] args)
    {
        string url = "www.stackoverflow.com";
        var pingTask = PingUrlAsync(url);
        Console.WriteLine($"Waiting for response from {url}");
        Task.WaitAll(pingTask);
        Console.WriteLine(pingTask.Result);
    }

    static async Task<string> PingUrlAsync(string url)
    {
        string response = string.Empty;
        var ping = new System.Net.NetworkInformation.Ping();

        var result = await ping.SendPingAsync(url);

        await Task.Delay(5000); //simulate slow internet

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            response = $"{url} is online";
        }

        return response;
    }
```

Leggi Task Libreria parallela online: https://riptutorial.com/it/csharp/topic/1010/task-libreria-parallela

Capitolo 150: threading

Osservazioni

Un **thread** è una parte di un programma che può essere eseguito indipendentemente dalle altre parti. Può eseguire attività contemporaneamente con altri thread. **Il multithreading** è una funzionalità che consente ai programmi di eseguire l'elaborazione simultanea in modo che sia possibile eseguire più di una operazione alla volta.

Ad esempio, è possibile utilizzare il threading per aggiornare un timer o un contatore in background mentre si eseguono contemporaneamente altre attività in primo piano.

Le applicazioni multithread sono più reattive all'input dell'utente e sono anche facilmente scalabili, poiché lo sviluppatore può aggiungere thread come e quando il carico di lavoro aumenta.

Per impostazione predefinita, un programma C # ha un thread, il thread del programma principale. Tuttavia, i thread secondari possono essere creati e utilizzati per eseguire codice in parallelo con il thread principale. Tali thread sono chiamati thread di lavoro.

Per controllare l'operazione di un thread, CLR delega una funzione al sistema operativo noto come Thread Scheduler. Uno scheduler di thread assicura che tutti i thread siano assegnati al tempo di esecuzione corretto. Controlla inoltre che i thread bloccati o bloccati non consumino gran parte del tempo di CPU.

.NET Framework `System.Threading` namespace semplifica l'utilizzo dei thread. System.Threading consente il multithreading fornendo un numero di classi e interfacce. Oltre a fornire tipi e classi per un particolare thread, definisce anche i tipi per contenere una collezione di thread, una classe timer e così via. Fornisce inoltre il suo supporto consentendo l'accesso sincronizzato ai dati condivisi.

`Thread` è la classe principale nello spazio dei nomi `System.Threading` . Altre classi includono `AutoResetEvent` , `Interlocked` , `Monitor` , `Mutex` e `ThreadPool` .

Alcuni dei delegati presenti nello spazio dei nomi `System.Threading` includono `ThreadStart` , `TimerCallback` e `WaitCallback` .

Le enumerazioni nello spazio dei nomi `System.Threading` includono `ThreadPriority` , `ThreadState` e `EventResetMode` .

In .NET Framework 4 e versioni successive, la programmazione multithreading è resa più semplice e più semplice attraverso le classi `System.Threading.Tasks.Parallel` e `System.Threading.Tasks.Task` , Parallel LINQ (PLINQ), nuove classi di raccolta simultanee in `System.Collections.Concurrent` nomi `System.Collections.Concurrent` e un nuovo modello di programmazione basato sulle attività.

Examples

Semplice demo di threading completa

```
class Program
{
    static void Main(string[] args)
    {
        // Create 2 thread objects.  We're using delegates because we need to pass
        // parameters to the threads.
        var thread1 = new Thread(new ThreadStart(() => PerformAction(1)));
        var thread2 = new Thread(new ThreadStart(() => PerformAction(2)));

        // Start the threads running
        thread1.Start();
        // NB: as soon as the above line kicks off the thread, the next line starts;
        // even if thread1 is still processing.
        thread2.Start();

        // Wait for thread1 to complete before continuing
        thread1.Join();
        // Wait for thread2 to complete before continuing
        thread2.Join();

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Thread: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

Demo di threading completo semplice tramite l'attività

```
class Program
{
    static void Main(string[] args)
    {
        // Run 2 Tasks.
        var task1 = Task.Run(() => PerformAction(1));
        var task2 = Task.Run(() => PerformAction(2));

        // Wait (i.e. block this thread) until both Tasks are complete.
        Task.WaitAll(new [] { task1, task2 });

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
```

```
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Task: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

Esplicito compito Parallism

```
    private static void explicitTaskParallism()
    {
        Thread.CurrentThread.Name = "Main";

        // Create a task and supply a user delegate by using a lambda expression.
        Task taskA = new Task(() => Console.WriteLine($"Hello from task {nameof(taskA)}."));
        Task taskB = new Task(() => Console.WriteLine($"Hello from task {nameof(taskB)}."));

        // Start the task.
        taskA.Start();
        taskB.Start();

        // Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name);
        taskA.Wait();
        taskB.Wait();
        Console.Read();
    }
```

Parallelismo di attività implicite

```
    private static void Main(string[] args)
    {
        var a = new A();
        var b = new B();
        //implicit task parallelism
        Parallel.Invoke(
            () => a.DoSomeWork(),
            () => b.DoSomeOtherWork()
            );

      }
```

Creazione e avvio di una seconda discussione

Se stai eseguendo più calcoli lunghi, puoi eseguirli contemporaneamente su diversi thread sul tuo computer. Per fare ciò, creiamo un nuovo **Thread** e puntiamo a un metodo diverso.

```
using System.Threading;

class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
```

```
        thread.Start();
    }

    static void Secondary() {
        System.Console.WriteLine("Hello World!");
    }
}
```

Avvio di una discussione con parametri

using System.Threading;

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start("SecondThread");
    }

    static void Secondary(object threadName) {
        System.Console.WriteLine("Hello World from thread: " + threadName);
    }
}
```

Creazione di un thread per processore

> `Environment.ProcessorCount` Ottiene il numero di processori **logici** sulla macchina corrente.

Il CLR pianificherà quindi ogni thread su un processore logico, questo teoricamente potrebbe significare ogni thread su un processore logico diverso, tutti i thread su un singolo processore logico o qualche altra combinazione.

```
using System;
using System.Threading;

class MainClass {
    static void Main() {
        for (int i = 0; i < Environment.ProcessorCount; i++) {
            var thread = new Thread(Secondary);
            thread.Start(i);
        }

    }

    static void Secondary(object threadNumber) {
        System.Console.WriteLine("Hello World from thread: " + threadNumber);
    }
}
```

Evitare di leggere e scrivere dati contemporaneamente

A volte, vuoi che i tuoi thread condividano simultaneamente i dati. Quando ciò accade, è importante conoscere il codice e bloccare le parti che potrebbero andare storte. Di seguito viene

mostrato un semplice esempio di conteggio di due thread.

Ecco un codice pericoloso (errato):

```
using System.Threading;

class MainClass
{
    static int count { get; set; }

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            var temp = count;
            System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
            Thread.Sleep(1000);
            count = temp + 1;
            System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of
count to:" + count);
            Thread.Sleep(1000);
        }
    }
}
```

Noterai, invece di contare 1,2,3,4,5 ... contiamo 1,1,2,2,3 ...

Per risolvere questo problema, dobbiamo **bloccare** il valore del conteggio, in modo che più thread diversi non possano leggere e scrivere allo stesso tempo. Con l'aggiunta di un blocco e una chiave, possiamo impedire ai thread di accedere ai dati simultaneamente.

```
using System.Threading;

class MainClass
{

    static int count { get; set; }
    static readonly object key = new object();

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
```

```
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            lock (key)
            {
                var temp = count;
                System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
                Thread.Sleep(1000);
                count = temp + 1;
                System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value
of count to:" + count);
            }
            Thread.Sleep(1000);
        }
    }
}
```

Parallelo. Per ogni ciclo

Se si dispone di un ciclo foreach che si desidera velocizzare e non si preoccupa di quale ordine è l'output, è possibile convertirlo in un ciclo foreach parallelo attenendosi alla seguente procedura:

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class MainClass {

    public static void Main() {
        int[] Numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // Single-threaded
        Console.WriteLine("Normal foreach loop: ");
        foreach (var number in Numbers) {
            Console.WriteLine(longCalculation(number));
        }
        // This is the Parallel (Multi-threaded solution)
        Console.WriteLine("Parallel foreach loop: ");
        Parallel.ForEach(Numbers, number => {
            Console.WriteLine(longCalculation(number));
        });
    }

    private static int longCalculation(int number) {
        Thread.Sleep(1000); // Sleep to simulate a long calculation
        return number * number;
    }
}
```

Deadlock (due thread in attesa l'uno dell'altro)

Un deadlock è ciò che si verifica quando due o più thread sono in attesa di completamento o di rilascio di una risorsa in modo tale da attendere per sempre.

Uno scenario tipico di due thread in attesa di completamento è quando un thread GUI di Windows Form attende un thread di lavoro e il thread worker tenta di richiamare un oggetto gestito dal thread della GUI. Osservare che con questo codice exmaple, facendo clic su button1 si bloccherà il programma.

```
private void button1_Click(object sender, EventArgs e)
{
    Thread workerthread= new Thread(dowork);
    workerthread.Start();
    workerthread.Join();
    // Do something after
}

private void dowork()
{
    // Do something before
    textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
    // Do something after
}
```

`workerthread.Join()` è una chiamata che blocca il thread chiamante fino al completamento di workthread. `textBox1.Invoke(invoke_delegate)` è una chiamata che blocca il thread chiamante finché il thread della GUI non ha elaborato invoke_delegate, ma questa chiamata provoca deadlock se il thread della GUI è già in attesa del completamento del thread chiamante.

Per ovviare a questo, si può usare un modo non bloccante di invocare la casella di testo invece:

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => textBox1.Text = "Some Text"));
    // Do work that is not dependent on textBox1 being updated first
}
```

Tuttavia, ciò causerà problemi se è necessario eseguire il codice che dipende dalla casella di testo che viene aggiornata per prima. In tal caso, eseguilo come parte del richiamo, ma tieni presente che questo lo farà girare sul thread della GUI.

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => {
        textBox1.Text = "Some Text";
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    }));
    // Do work that is not dependent on textBox1 being updated first
}
```

In alternativa, avvia un nuovo thread intero e lascia che quello faccia l'attesa sul thread della GUI, in modo che workthread possa essere completato.

```
private void dowork()
```

```
{
    // Do work
    Thread workerthread2 = new Thread(() =>
    {
        textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    });
    workerthread2.Start();
    // Do work that is not dependent on textBox1 being updated first
}
```

Per ridurre al minimo il rischio di imbattersi in un deadlock di attesa reciproca, evitare sempre i riferimenti circolari tra i thread quando possibile. Una gerarchia di thread in cui i thread di livello inferiore lasciano solo i messaggi per i thread di livello superiore e non li attendono mai non si imbatteranno in questo tipo di problema. Tuttavia, sarebbe comunque vulnerabile ai deadlock basati sul blocco delle risorse.

Deadlock (mantieni la risorsa e aspetta)

Un deadlock è ciò che si verifica quando due o più thread sono in attesa di completamento o di rilascio di una risorsa in modo tale da attendere per sempre.

Se thread1 contiene un blocco sulla risorsa A e attende che la risorsa B venga rilasciata mentre thread2 contiene la risorsa B e attende che la risorsa A venga rilasciata, sono bloccati.

Facendo clic sul pulsante 1 per il seguente codice di esempio, la tua applicazione entrerà in uno stato di stallo imprevisto e si bloccherà

```
private void button_Click(object sender, EventArgs e)
{
    DeadlockWorkers workers = new DeadlockWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class DeadlockWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;

    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
```

```
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
        Thread.Sleep(100);
        lock (resourceA)
        {
            Thread.Sleep(100);
            lock (resourceB)
            {
                output += "T1#";
            }
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}
```

Per evitare di essere bloccati in questo modo, è possibile utilizzare Monitor.TryEnter (lock_object, timeout_in_milliseconds) per verificare se un blocco è già presente su un oggetto. Se Monitor.TryEnter non riesce ad acquisire un lock su lock_object prima di timeout_in_milliseconds, restituisce false, dando al thread la possibilità di rilasciare altre risorse detenute e cedendo, dando così agli altri thread la possibilità di completare come in questa versione leggermente modificata di quanto sopra :

```
private void button_Click(object sender, EventArgs e)
{
    MonitorWorkers workers = new MonitorWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class MonitorWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;

    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
```

```
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
        bool mustDoWork = true;
        Thread.Sleep(100);
        while (mustDoWork)
        {
            lock (resourceA)
            {
                Thread.Sleep(100);
                if (Monitor.TryEnter(resourceB, 0))
                {
                    output += "T1#";
                    mustDoWork = false;
                    Monitor.Exit(resourceB);
                }
            }
            if (mustDoWork) Thread.Yield();
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}
```

Si noti che questa soluzione alternativa si basa sul fatto che thread2 è testardo sui blocchi e che thread1 è disposto a fornire, in modo che thread2 abbia sempre la precedenza. Si noti inoltre che thread1 deve rifare il lavoro che ha svolto dopo aver bloccato la risorsa A, quando produce. Pertanto, sii cauto nell'implementare questo approccio con più di un thread di rendimento, poiché rischierai di entrare in un cosiddetto livelock - uno stato che si verificherebbe se due thread continuassero a fare il primo bit del loro lavoro e poi si cedere reciprocamente , ricominciare più volte.

Leggi threading online: https://riptutorial.com/it/csharp/topic/51/threading

Capitolo 151: Timer

Sintassi

- `myTimer.Interval` - imposta la frequenza con cui viene chiamato l'evento "Tick" (in millisecondi)
- `myTimer.Enabled` - valore booleano che imposta il timer su abilitato / disabilitato
- `myTimer.Start()` - Avvia il timer.
- `myTimer.Stop()` - Ferma il timer.

Osservazioni

Se si utilizza Visual Studio, i timer possono essere aggiunti come controllo direttamente al modulo dalla casella degli strumenti.

Examples

Timer multithread

`System.Threading.Timer` - Timer multithreading più semplice. Contiene due metodi e un costruttore.

Esempio: un timer chiama il metodo DataWrite, che scrive "multithread eseguito ..." dopo che sono trascorsi cinque secondi, e quindi ogni secondo dopo che l'utente preme Invio:

```
using System;
using System.Threading;
class Program
{
  static void Main()
  {
    // First interval = 5000ms; subsequent intervals = 1000ms
    Timer timer = new Timer (DataWrite, "multithread executed...", 5000, 1000);
    Console.ReadLine();
    timer.Dispose(); // This both stops the timer and cleans up.
  }

  static void DataWrite (object data)
  {
    // This runs on a pooled thread
    Console.WriteLine (data); // Writes "multithread executed..."
  }
}
```

Nota: pubblicherà una sezione separata per lo smaltimento dei timer multithread.

`Change` : questo metodo può essere chiamato quando si desidera modificare l'intervallo del timer.

`Timeout.Infinite` - Se vuoi sparare solo una volta. Specificare questo nell'ultimo argomento del costruttore.

`System.Timers` - Un'altra classe di timer fornita da .NET Framework. Termina `System.Threading.Timer` .

Caratteristiche:

- `IComponent` : consente di posizionarlo nella barra dei componenti del Designer di Visual Studio
- Proprietà `Interval` invece di un metodo `Change`
- `event Elapsed` anziché un `delegate` richiamata
- Proprietà `Enabled` per avviare e arrestare il timer (`default value = false`)
- `Start` e `Stop` metodi nel caso in cui ti venga confusa la proprietà `Enabled` (punto precedente)
- `AutoReset` - per indicare un evento ricorrente (`default value = true`)
- Proprietà `SynchronizingObject` con i metodi `Invoke` e `BeginInvoke` per chiamare in sicurezza i metodi su elementi WPF e controlli Windows Form

Esempio che rappresenta tutte le caratteristiche di cui sopra:

```
using System;
using System.Timers; // Timers namespace rather than Threading
class SystemTimer
{
  static void Main()
  {
    Timer timer = new Timer(); // Doesn't require any args
    timer.Interval = 500;
    timer.Elapsed += timer_Elapsed; // Uses an event instead of a delegate
    timer.Start(); // Start the timer
    Console.ReadLine();
    timer.Stop(); // Stop the timer
    Console.ReadLine();
    timer.Start(); // Restart the timer
    Console.ReadLine();
    timer.Dispose(); // Permanently stop the timer
 }

 static void timer_Elapsed(object sender, EventArgs e)
 {
   Console.WriteLine ("Tick");
 }
}
```

`Multithreaded timers` : utilizzare il pool di thread per consentire a pochi thread di servire molti timer. Significa che il metodo di callback o l'evento `Elapsed` può attivarsi su un thread diverso ogni volta che viene chiamato.

`Elapsed` : questo evento si attiva sempre in tempo, indipendentemente dal fatto che l'evento `Elapsed` precedente abbia completato l'esecuzione. Per questo motivo, callback o gestori di eventi devono essere thread-safe. La precisione dei timer multithread dipende dal sistema operativo ed è in genere compresa tra 10 e 20 ms.

`interop` - quando hai bisogno di maggiore precisione, usa questo e chiama il timer multimediale di Windows. Questo ha una precisione fino a 1 ms ed è definito in `winmm.dll` .

`timeBeginPeriod` : chiamate prima questo per informare il sistema operativo che è necessaria un'accuratezza dei tempi elevata

`timeSetEvent` - chiama questo dopo il `timeBeginPeriod` di `timeBeginPeriod` per avviare un timer multimediale.

`timeKillEvent` : chiama questo quando hai finito, questo arresta il timer

`timeEndPeriod` - Chiama questo per informare l'OS che non hai più bisogno di un'accuratezza dei tempi elevata.

Puoi trovare esempi completi su Internet che utilizzano il timer multimediale cercando le parole chiave `dllimport winmm.dll timesetevent` .

Creazione di un'istanza di un timer

I timer sono utilizzati per eseguire attività a intervalli di tempo specifici (Do X ogni Y secondi) Di seguito è riportato un esempio di creazione di una nuova istanza di un Timer.

NOTA : questo vale per i timer che utilizzano WinForms. Se si utilizza WPF, è possibile esaminare `DispatcherTimer`

```
    using System.Windows.Forms; //Timers use the Windows.Forms namespace

    public partial class Form1 : Form
    {

        Timer myTimer = new Timer(); //create an instance of Timer named myTimer


        public Form1()
        {
            InitializeComponent();
        }

    }
```

Assegnazione del gestore di eventi "Tick" a un Timer

Tutte le azioni eseguite in un timer vengono gestite nell'evento "Tick".

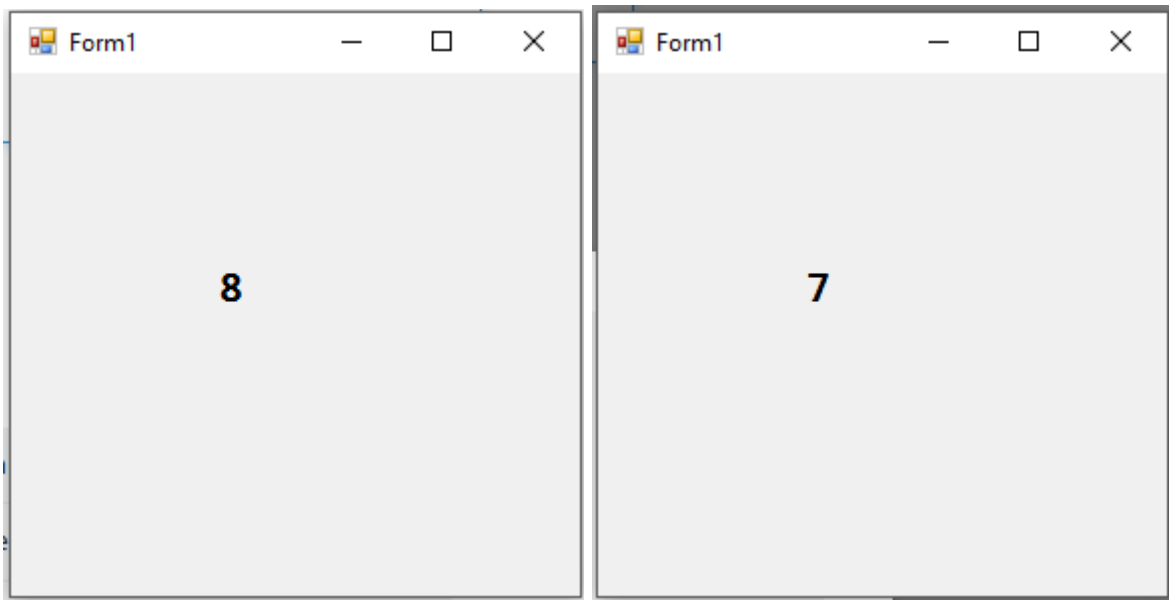
```
public partial class Form1 : Form
{

    Timer myTimer = new Timer();


    public Form1()
    {
        InitializeComponent();

        myTimer.Tick += myTimer_Tick; //assign the event handler named "myTimer_Tick"
    }
```

```
    private void myTimer_Tick(object sender, EventArgs e)
    {
        // Perform your actions here.
    }
}
```

Esempio: utilizzo di un timer per eseguire un semplice conto alla rovescia.

```
    public partial class Form1 : Form
    {

    Timer myTimer = new Timer();
    int timeLeft = 10;

        public Form1()
        {
            InitializeComponent();

            //set properties for the Timer
            myTimer.Interval = 1000;
            myTimer.Enabled = true;

            //Set the event handler for the timer, named "myTimer_Tick"
            myTimer.Tick += myTimer_Tick;

            //Start the timer as soon as the form is loaded
            myTimer.Start();

            //Show the time set in the "timeLeft" variable
            lblCountDown.Text = timeLeft.ToString();

        }

        private void myTimer_Tick(object sender, EventArgs e)
        {
            //perform these actions at the interval set in the properties.
            lblCountDown.Text = timeLeft.ToString();
            timeLeft -= 1;

            if (timeLeft < 0)
            {
                myTimer.Stop();
            }
        }
    }
```

Risultati in ...



E così via...

Leggi Timer online: https://riptutorial.com/it/csharp/topic/3829/timer

Capitolo 152: Tipi anonimi

Examples

Creare un tipo anonimo

Poiché i tipi anonimi non vengono denominati, le variabili di questi tipi devono essere digitate implicitamente (`var`).

```
var anon = new { Foo = 1, Bar = 2 };
// anon.Foo == 1
// anon.Bar == 2
```

Se i nomi dei membri non sono specificati, vengono impostati sul nome della proprietà / variabile utilizzata per inizializzare l'oggetto.

```
int foo = 1;
int bar = 2;
var anon2 = new { foo, bar };
// anon2.foo == 1
// anon2.bar == 2
```

Si noti che i nomi possono essere omessi solo quando l'espressione nella dichiarazione del tipo anonimo è un semplice accesso alla proprietà; per le chiamate ai metodi o le espressioni più complesse, è necessario specificare un nome di proprietà.

```
string foo = "some string";
var anon3 = new { foo.Length };
// anon3.Length == 11
var anon4 = new { foo.Length <= 10 ? "short string" : "long string" };
// compiler error - Invalid anonymous type member declarator.
var anon5 = new { Description = foo.Length <= 10 ? "short string" : "long string" };
// OK
```

Anonimo vs dinamico

I tipi anonimi consentono la creazione di oggetti senza dover definire esplicitamente i loro tipi in anticipo, mantenendo il controllo di tipo statico.

```
var anon = new { Value = 1 };
Console.WriteLine(anon.Id); // compile time error
```

Al contrario, `dynamic` ha il controllo dinamico dei tipi, optando per gli errori di runtime, invece degli errori di compilazione.

```
dynamic val = "foo";
Console.WriteLine(val.Id); // compiles, but throws runtime error
```

Metodi generici con tipi anonimi

I metodi generici consentono l'uso di tipi anonimi attraverso l'inferenza di tipo.

```
void Log<T>(T obj) {
    // ...
}
Log(new { Value = 10 });
```

Ciò significa che le espressioni LINQ possono essere utilizzate con tipi anonimi:

```
var products = new[] {
    new { Amount = 10, Id = 0 },
    new { Amount = 20, Id = 1 },
    new { Amount = 15, Id = 2 }
};
var idsByAmount = products.OrderBy(x => x.Amount).Select(x => x.Id);
// idsByAmount: 0, 2, 1
```

Creazione di istanze di tipi generici con tipi anonimi

L'utilizzo di costruttori generici richiederebbe il nome dei tipi anonimi, il che non è possibile. In alternativa, è possibile utilizzare metodi generici per consentire l'inferenza del tipo.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 5, Bar = 10 };
List<T> CreateList<T>(params T[] items) {
    return new List<T>(items);
}

var list1 = CreateList(anon, anon2);
```

Nel caso di `List<T>` , gli array implicitamente tipizzati possono essere convertiti in un `List<T>` tramite il metodo LINQ `ToList` :

```
var list2 = new[] {anon, anon2}.ToList();
```

Uguaglianza di tipo anonimo

L'uguaglianza di tipo anonimo è data dal metodo di istanza `Equals` . Due oggetti sono uguali se hanno lo stesso tipo e valori uguali (attraverso `a.Prop.Equals(b.Prop)`) per ogni proprietà.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 1, Bar = 2 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon4 = new { Bar = 2, Foo = 1 };
// anon.Equals(anon2) == true
// anon.Equals(anon3) == false
// anon.Equals(anon4) == false (anon and anon4 have different types, see below)
```

Due tipi anonimi sono considerati uguali se e solo se le loro proprietà hanno lo stesso nome e tipo e appaiono nello stesso ordine.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 7, Bar = 1 };
var anon3 = new { Bar = 1, Foo = 3 };
var anon4 = new { Fa = 1, Bar = 2 };
// anon and anon2 have the same type
// anon and anon3 have diferent types (Bar and Foo appear in different orders)
// anon and anon4 have different types (property names are different)
```

Array implicitamente tipizzati

Le matrici di tipi anonimi possono essere create con una digitazione implicita.

```
var arr = new[] {
    new { Id = 0 },
    new { Id = 1 }
};
```

Leggi Tipi anonimi online: https://riptutorial.com/it/csharp/topic/765/tipi-anonimi

Capitolo 153: Tipi incorporati

Examples

Tipo di riferimento immutabile - stringa

```
// assign string from a string literal
string s = "hello";

// assign string from an array of characters
char[] chars = new char[] { 'h', 'e', 'l', 'l', 'o' };
string s = new string(chars, 0, chars.Length);

// assign string from a char pointer, derived from a string
string s;
unsafe
{
    fixed (char* charPointer = "hello")
    {
        s = new string(charPointer);
    }
}
```

Tipo di valore - char

```
// single character s
char c = 's';

// character s: casted from integer value
char c = (char)115;

// unicode character: single character s
char c = '\u0073';

// unicode character: smiley face
char c = '\u263a';
```

Tipo di valore: breve, int, lungo (con segno 16 bit, 32 bit, numeri interi a 64 bit)

```
// assigning a signed short to its minimum value
short s = -32768;

// assigning a signed short to its maximum value
short s = 32767;

// assigning a signed int to its minimum value
int i = -2147483648;

// assigning a signed int to its maximum value
int i = 2147483647;

// assigning a signed long to its minimum value (note the long postfix)
long l = -9223372036854775808L;
```

```
// assigning a signed long to its maximum value (note the long postfix)
long l = 9223372036854775807L;
```

È anche possibile rendere nullable di questi tipi, il che significa che in aggiunta ai normali valori può essere assegnato anche null. Se una variabile di un tipo nullable non è inizializzata, sarà null invece di 0. I tipi di nullità sono contrassegnati aggiungendo un punto interrogativo (?) Dopo il tipo.

```
int a; //This is now 0.
int? b; //This is now null.
```

Tipo di valore: ushort, uint, ulong (interi senza segno a 16 bit, 32 bit, 64 bit)

```
// assigning an unsigned short to its minimum value
ushort s = 0;

// assigning an unsigned short to its maximum value
ushort s = 65535;

// assigning an unsigned int to its minimum value
uint i = 0;

// assigning an unsigned int to its maximum value
uint i = 4294967295;

// assigning an unsigned long to its minimum value (note the unsigned long postfix)
ulong l = 0UL;

// assigning an unsigned long to its maximum value (note the unsigned long postfix)
ulong l = 18446744073709551615UL;
```

È anche possibile rendere nullable di questi tipi, il che significa che in aggiunta ai normali valori può essere assegnato anche null. Se una variabile di un tipo nullable non è inizializzata, sarà null invece di 0. I tipi di nullità sono contrassegnati aggiungendo un punto interrogativo (?) Dopo il tipo.

```
uint a; //This is now 0.
uint? b; //This is now null.
```

Tipo di valore - bool

```
// default value of boolean is false
bool b;
//default value of nullable boolean is null
bool? z;
b = true;
if(b) {
    Console.WriteLine("Boolean has true value");
}
```

La parola chiave bool è un alias di System.Boolean. È usato per dichiarare le variabili per memorizzare i valori booleani, `true` e `false` .

Confronti con i tipi di valore in scatola

Se i tipi di valore sono assegnati a variabili di tipo `object` , vengono *inseriti* in una *scatola* : il valore viene archiviato in un'istanza di `System.Object` . Questo può portare a conseguenze indesiderate quando si confrontano i valori con `==` , ad esempio:

```
object left = (int)1;  // int in an object box
object right = (int)1; // int in an object box

var comparison1 = left == right;      // false
```

Questo può essere evitato usando il metodo `Equals` sovraccarico, che darà il risultato atteso.

```
var comparison2 = left.Equals(right); // true
```

In alternativa, lo stesso potrebbe essere fatto unboxing delle variabili `left` e `right` modo che i valori `int` vengano confrontati:

```
var comparison3 = (int)left == (int)right; // true
```

Conversione di tipi di valore in box

I tipi di valore in box possono essere disgiunti solo nel loro `Type` originale, anche se una conversione dei due `Type` è valida, ad esempio:

```
object boxedInt = (int)1; // int boxed in an object

long unboxedInt1 = (long)boxedInt; // invalid cast
```

Questo può essere evitato con il primo unboxing nel `Type` originale, ad esempio:

```
long unboxedInt2 = (long)(int)boxedInt; // valid
```

Leggi Tipi incorporati online: https://riptutorial.com/it/csharp/topic/42/tipi-incorporati

Capitolo 154: Tipo di conversione

Osservazioni

La conversione del tipo sta convertendo un tipo di dati in un altro tipo. È anche noto come tipo di fusione. In C #, il casting del tipo ha due forme:

Conversione implicita del tipo : queste conversioni vengono eseguite da C # in modo sicuro per i tipi. Ad esempio, sono le conversioni da tipi interi e conversioni da più piccoli a più grandi da classi derivate a classi base.

Conversione esplicita del tipo : queste conversioni vengono eseguite esplicitamente dagli utenti che utilizzano le funzioni predefinite. Le conversioni esplicite richiedono un operatore di cast.

Examples

Esempio di operatore implicito MSDN

```
class Digit
{
    public Digit(double d) { val = d; }
    public double val;

    // User-defined conversion from Digit to double
    public static implicit operator double(Digit d)
    {
        Console.WriteLine("Digit to double implict conversion called");
        return d.val;
    }
    //  User-defined conversion from double to Digit
    public static implicit operator Digit(double d)
    {
        Console.WriteLine("double to Digit implict conversion called");
        return new Digit(d);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        //This call invokes the implicit "double" operator
        double num = dig;
        //This call invokes the implicit "Digit" operator
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

Produzione:

Digita per raddoppiare la conversione implicita chiamata
Conversazione implicita da doppia a cifra chiamata
num = 7 dig2 = 12

Live Demo su .NET Fiddle

Conversione di tipo esplicita

```
using System;
namespace TypeConversionApplication
{
   class ExplicitConversion
   {
      static void Main(string[] args)
      {
         double d = 5673.74;
         int i;

         // cast double to int.
         i = (int)d;
         Console.WriteLine(i);
         Console.ReadKey();
      }
   }
}
```

Leggi Tipo di conversione online: https://riptutorial.com/it/csharp/topic/3489/tipo-di-conversione

Capitolo 155: Tipo di valore vs Tipo di riferimento

Sintassi

- Passando per riferimento: public void Double (ref int numberToDouble) {}

Osservazioni

introduzione

Tipi di valore

I tipi di valore sono i più semplici dei due. I tipi di valore sono spesso usati per rappresentare i dati stessi. Un numero intero, un punto booleano o un punto nello spazio 3D sono tutti esempi di buoni valori.

I tipi di valore (structs) sono dichiarati usando la parola chiave struct. Vedere la sezione della sintassi per un esempio su come dichiarare una nuova struttura.

In generale, abbiamo 2 parole chiave che vengono utilizzate per dichiarare i tipi di valore:

- Structs
- enumerazioni

Tipi di riferimento

I tipi di riferimento sono leggermente più complessi. I tipi di riferimento sono oggetti tradizionali nel senso della programmazione orientata agli oggetti. Quindi, supportano l'ereditarietà (e i benefici che ne derivano) e supportano anche i finalizzatori.

In C # generalmente abbiamo questo tipo di riferimento:

- Classi
- I delegati
- interfacce

Nuovi tipi di riferimento (classi) sono dichiarati usando la parola chiave class. Per un esempio, vedere la sezione della sintassi su come dichiarare un nuovo tipo di riferimento.

Major Differences

Di seguito sono riportate le principali differenze tra tipi di riferimento e tipi di valore.

Esistono tipi di valore nello stack, esistono tipi di riferimento sull'heap

Questa è la differenza spesso menzionata tra i due, ma in realtà, ciò a cui si riduce è che quando si utilizza un tipo di valore in C #, ad esempio un int, il programma utilizzerà quella variabile per fare riferimento direttamente a quel valore. Se dici int mine = 0, la variabile mine si riferisce direttamente a 0, che è efficiente. Tuttavia, i tipi di riferimento effettivamente contengono (come suggerisce il nome) un riferimento all'oggetto sottostante, questo è simile ai puntatori in altri linguaggi come C ++.

Potresti non notare immediatamente gli effetti di questo, ma gli effetti sono lì, sono potenti e sono sottili. Vedere l'esempio sulla modifica dei tipi di riferimento altrove per un esempio.

Questa differenza è la ragione principale per le seguenti altre differenze, e vale la pena conoscere.

I tipi di valore non cambiano quando li si cambia in un metodo, i tipi di riferimento lo fanno

Quando un tipo di valore viene passato in un metodo come parametro, se il metodo modifica il valore in qualsiasi modo, il valore non viene modificato Al contrario, passando un tipo di riferimento nello stesso metodo e cambiandolo si cambia l'oggetto sottostante, in modo che altre cose che usano lo stesso oggetto avranno l'oggetto appena cambiato piuttosto che il loro valore originale.

Vedi l'esempio dei tipi di valore rispetto ai tipi di riferimento nei metodi per maggiori informazioni.

Cosa succede se voglio cambiarli?

Semplicemente passali nel tuo metodo usando la parola chiave "ref", e quindi stai passando questo oggetto per riferimento. Significato, è lo stesso oggetto in memoria. Quindi le modifiche che apporti saranno rispettate. Vedere l'esempio sul passaggio per riferimento per un esempio.

I tipi di valore non possono essere nulli, i tipi di riferimento possono

Praticamente come si dice, è possibile assegnare un valore nullo a un tipo di riferimento, nel senso che alla variabile assegnata non può essere assegnato alcun oggetto reale. Nel caso di tipi di valore, tuttavia, ciò non è possibile. Tuttavia, puoi usare Nullable, per consentire al tuo tipo di valore di essere annullabile, se questo è un requisito, anche se questo è qualcosa che stai considerando, pensa fortemente se una classe potrebbe non essere l'approccio migliore qui, se è il tuo genere.

Examples

Modifica dei valori altrove

```
public static void Main(string[] args)
{
```

```
    var studentList = new List<Student>();
    studentList.Add(new Student("Scott", "Nuke"));
    studentList.Add(new Student("Vincent", "King"));
    studentList.Add(new Student("Craig", "Bertt"));

    // make a separate list to print out later
    var printingList = studentList; // this is a new list object, but holding the same student
objects inside it

    // oops, we've noticed typos in the names, so we fix those
    studentList[0].LastName = "Duke";
    studentList[1].LastName = "Kong";
    studentList[2].LastName = "Brett";

    // okay, we now print the list
    PrintPrintingList(printingList);
}

private static void PrintPrintingList(List<Student> students)
{
    foreach (Student student in students)
    {
        Console.WriteLine(string.Format("{0} {1}", student.FirstName, student.LastName));
    }
}
```

Si noterà che anche se l'elenco delle liste di stampa è stato creato prima delle correzioni ai nomi degli studenti dopo gli errori di battitura, il metodo PrintPrintingList stampa ancora i nomi corretti:

```
Scott Duke
Vincent Kong
Craig Brett
```

Questo perché entrambe le liste contengono un elenco di riferimenti agli stessi studenti. COSÌ, cambiando l'oggetto studente sottostante si propaga agli usi da entrambe le liste.

Ecco come sarà la classe studentesca.

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Student(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

Passando per riferimento

Se si desidera che i tipi di valore e i tipi di riferimento nell'esempio di metodi funzionino correttamente, utilizzare la parola chiave ref nella firma del metodo per il parametro che si desidera passare per riferimento, nonché quando si chiama il metodo.

```
public static void Main(string[] args)
{
    ...
    DoubleNumber(ref number); // calling code
    Console.WriteLine(number); // outputs 8
    ...
}
```

```
public void DoubleNumber(ref int number)
{
    number += number;
}
```

Apportare queste modifiche renderebbe il numero di aggiornamento come previsto, il che significa che l'output della console per il numero sarebbe 8.

Passando per riferimento usando la parola chiave ref.

Dalla documentazione :

> In C #, gli argomenti possono essere passati ai parametri per valore o per riferimento. Il passaggio per riferimento abilita membri di funzioni, metodi, proprietà, indicizzatori, operatori e costruttori per modificare il valore dei parametri e mantenere tale modifica nell'ambiente di chiamata. Per passare un parametro per riferimento, utilizzare il `ref` o `out` parola chiave.

La differenza tra `ref` e `out` è che `out` significa che il parametro passato deve essere assegnato prima della fine della funzione. I parametri di contrasto passati con `ref` possono essere modificati o lasciati invariati.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);

        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a = 5;
```

```
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a = 6;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        a = 7;
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

Uscita :

```
Inside Main - Before Callee: a = 20
Inside Callee a : 5
Inside Main - After Callee: a = 20
Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 6
Inside Main - After CalleeRef: a = 6
Inside Main - Before CalleeOut: a = 6
Inside CalleeOut a : 7
Inside Main - After CalleeOut: a = 7
```

assegnazione

```
var a = new List<int>();
var b = a;
a.Add(5);
Console.WriteLine(a.Count); // prints 1
Console.WriteLine(b.Count); // prints 1 as well
```

Assegnare a una variabile di una `List<int>` non crea una copia della `List<int>` . Invece, copia il riferimento alla `List<int>` . Chiamiamo tipi che si comportano in questo modo *tipi di riferimento* .

Differenza con i parametri del metodo ref e out

Esistono due modi possibili per passare un tipo di valore per riferimento: `ref` e `out` . La differenza è che passandolo con `ref` il valore deve essere inizializzato ma non quando lo si passa `out` .
L'utilizzo di `out` assicura che la variabile abbia un valore dopo la chiamata al metodo:

```
public void ByRef(ref int value)
{
    Console.WriteLine(nameof(ByRef) + value);
    value += 4;
    Console.WriteLine(nameof(ByRef) + value);
}

public void ByOut(out int value)
{
```

```
    value += 4 // CS0269: Use of unassigned out parameter `value'
    Console.WriteLine(nameof(ByOut) + value); // CS0269: Use of unassigned out parameter
`value'

    value = 4;
    Console.WriteLine(nameof(ByOut) + value);
}

public void TestOut()
{
    int outValue1;
    ByOut(out outValue1); // prints 4

    int outValue2 = 10;   // does not make any sense for out
    ByOut(out outValue2); // prints 4
}

public void TestRef()
{
    int refValue1;
    ByRef(ref refValue1); // S0165  Use of unassigned local variable 'refValue'

    int refValue2 = 0;
    ByRef(ref refValue2); // prints 0 and 4

    int refValue3 = 10;
    ByRef(ref refValue3); // prints 10 and 14
}
```

Il problema è che utilizzando `out` il parametro `must` essere inizializzato prima di lasciare il metodo, quindi il seguente metodo è possibile con `ref` ma non con `out` :

```
public void EmtyRef(bool condition, ref int value)
{
    if (condition)
    {
        value += 10;
    }
}

public void EmtyOut(bool condition, out int value)
{
    if (condition)
    {
        value = 10;
    }
} //CS0177: The out parameter 'value' must be assigned before control leaves the current
method
```

Questo perché se la `condition` non regge, il `value` non è assegnato.

parametri ref vs out

Codice

```
class Program
{
```

```
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);
        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a += 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a += 10;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        // can't use a+=15 since for this method 'a' is not intialized only declared in the
method declaration
        a = 25; //has to be initialized
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

Produzione

```
Inside Main - Before Callee: a = 20
Inside Callee a : 25
Inside Main - After Callee: a = 20

Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 30
Inside Main - After CalleeRef: a = 30

Inside Main - Before CalleeOut: a = 30
Inside CalleeOut a : 25
Inside Main - After CalleeOut: a = 25
```

Leggi Tipo di valore vs Tipo di riferimento online: https://riptutorial.com/it/csharp/topic/3014/tipo-di-valore-vs-tipo-di-riferimento

Capitolo 156: Tipo dinamico

Osservazioni

La parola chiave `dynamic` dichiara una variabile il cui tipo non è noto al momento della compilazione. Una variabile `dynamic` può contenere qualsiasi valore e il tipo di valore può cambiare durante il runtime.

Come notato nel libro "Metaprogramming in .NET", C # non ha un tipo di supporto per la parola chiave `dynamic` :

> La funzionalità abilitata dalla parola chiave `dynamic` è un insieme intelligente di azioni del compilatore che emettono e utilizzano oggetti `CallSite` nel contenitore del sito dell'ambito di esecuzione locale. Il compilatore gestisce ciò che i programmatori percepiscono come riferimenti di oggetti dinamici attraverso quelle istanze di `CallSite` . I parametri, i tipi di ritorno, i campi e le proprietà che ottengono il trattamento dinamico in fase di compilazione possono essere contrassegnati con alcuni metadati per indicare che sono stati generati per l'uso dinamico, ma il tipo di dati sottostante per loro sarà sempre `System.Object` .

Examples

Creare una variabile dinamica

```
dynamic foo = 123;
Console.WriteLine(foo + 234);
// 357    Console.WriteLine(foo.ToUpper())
// RuntimeBinderException, since int doesn't have a ToUpper method

foo = "123";
Console.WriteLine(foo + 234);
// 123234
Console.WriteLine(foo.ToUpper()):
// NOW A STRING
```

Tornando dinamico

```
using System;

public static void Main()
{
    var value = GetValue();
    Console.WriteLine(value);
    // dynamics are useful!
}

private static dynamic GetValue()
{
    return "dynamics are useful!";
```

```
}
```

Creazione di un oggetto dinamico con proprietà

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

Gestione di tipi specifici sconosciuti al momento della compilazione

I seguenti risultati equivalenti di output:

```
class IfElseExample
{
    public string DebugToString(object a)
    {
        if (a is StringBuilder)
        {
            return DebugToStringInternal(a as StringBuilder);
        }
        else if (a is List<string>)
        {
            return DebugToStringInternal(a as List<string>);
        }
        else
        {
            return a.ToString();
        }
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity},
Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}
```

```
class DynamicExample
{
    public string DebugToString(object a)
    {
        return DebugToStringInternal((dynamic)a);
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity},
Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}
```

Il vantaggio per la dinamica, è l'aggiunta di un nuovo tipo da gestire richiede solo l'aggiunta di un sovraccarico di DebugToStringInternal del nuovo tipo. Elimina anche la necessità di eseguirne manualmente il cast nel tipo.

Leggi Tipo dinamico online: https://riptutorial.com/it/csharp/topic/762/tipo-dinamico

Capitolo 157: Uguale e GetHashCode

Osservazioni

Ogni implementazione di `Equals` deve soddisfare i seguenti requisiti:

- **Riflessivo** : un oggetto deve essere uguale a se stesso.
  `x.Equals(x)` restituisce `true` .
- **Simmetrico** : non vi è alcuna differenza se si confronta x con y o y con x - il risultato è lo stesso.
  `x.Equals(y)` restituisce lo stesso valore di `y.Equals(x)` .
- **Transitivo** : se un oggetto è uguale a un altro oggetto e questo è uguale a un terzo, il primo deve essere uguale al terzo.
  if `(x.Equals(y) && y.Equals(z))` restituisce `true` , quindi `x.Equals(z)` restituisce `true` .
- **Coerentemente** : se si confronta un oggetto con un altro più volte, il risultato è sempre lo stesso.
  Le successive chiamate di `x.Equals(y)` restituiscono lo stesso valore fintanto che gli oggetti referenziati da x e y non vengono modificati.
- **Confronto con null** : nessun oggetto è uguale a `null` .
  `x.Equals(null)` restituisce `false` .

Implementazioni di `GetHashCode` :

- **Compatibile con `Equals`** : se due oggetti sono uguali (ovvero che `Equals` restituisce true), allora `GetHashCode` **deve** restituire lo stesso valore per ognuno di essi.
- **Ampio intervallo** : se due oggetti non sono uguali (`Equals` dice false), ci dovrebbe essere **un'alta probabilità che i** loro codici hash siano distinti. Spesso l'hashing *perfetto* non è possibile in quanto esiste un numero limitato di valori tra cui scegliere.
- **Economico** : dovrebbe essere poco costoso calcolare il codice hash in tutti i casi.

Vedi: Linee guida per il sovraccarico di Equals () e Operatore ==

Examples

Predefinito Comportamento uguale.

`Equals` è dichiarato nella classe `Object` stessa.

```
public virtual bool Equals(Object obj);
```

Per impostazione predefinita, `Equals` ha il seguente comportamento:

- Se l'istanza è un tipo di riferimento, `Equals` restituirà true solo se i riferimenti sono uguali.
- Se l'istanza è un tipo di valore, `Equals` restituirà true solo se il tipo e il valore sono uguali.
- `string` è un caso speciale. Si comporta come un tipo di valore.

```
namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            //areFooClassEqual: False
            Foo fooClass1 = new Foo("42");
            Foo fooClass2 = new Foo("42");
            bool areFooClassEqual = fooClass1.Equals(fooClass2);
            Console.WriteLine("fooClass1 and fooClass2 are equal: {0}", areFooClassEqual);
            //False

            //areFooIntEqual: True
            int fooInt1 = 42;
            int fooInt2 = 42;
            bool areFooIntEqual = fooInt1.Equals(fooInt2);
            Console.WriteLine("fooInt1 and fooInt2 are equal: {0}", areFooIntEqual);

            //areFooStringEqual: True
            string fooString1 = "42";
            string fooString2 = "42";
            bool areFooStringEqual = fooString1.Equals(fooString2);
            Console.WriteLine("fooString1 and fooString2 are equal: {0}", areFooStringEqual);
        }
    }

    public class Foo
    {
        public string Bar { get; }

        public Foo(string bar)
        {
            Bar = bar;
        }
    }
}
```

Scrittura di un buon override GetHashCode

`GetHashCode` ha importanti effetti sulle prestazioni del dizionario <> e HashTable.

Buoni metodi `GetHashCode`

- dovrebbe avere una distribuzione uniforme
  - ogni numero intero dovrebbe avere una probabilità all'incirca uguale di restituire per un'istanza casuale
  - se il tuo metodo restituisce lo stesso numero intero (ad es. la costante '999') per ogni istanza, avrai cattive prestazioni
- dovrebbe essere veloce

    - Questi NON sono hash crittografici, dove la lentezza è una caratteristica
    - più lenta è la tua funzione hash, più lento è il tuo dizionario
- deve restituire lo stesso HashCode su due istanze che `Equals` restituisce true
    - se non lo fanno (es. perché `GetHashCode` restituisce un numero casuale), gli elementi potrebbero non essere trovati in un `List` , `Dictionary` o simile.

Un buon metodo per implementare `GetHashCode` consiste nell'utilizzare un numero primo come valore iniziale e aggiungere gli hashcode dei campi del tipo moltiplicato per altri numeri primi a quello:

```
public override int GetHashCode()
{
    unchecked // Overflow is fine, just wrap
    {
        int hash = 3049; // Start value (prime number).

        // Suitable nullity checks etc, of course :)
        hash = hash * 5039 + field1.GetHashCode();
        hash = hash * 883 + field2.GetHashCode();
        hash = hash * 9719 + field3.GetHashCode();
        return hash;
    }
}
```

Solo i campi che sono usati in `Equals` -method dovrebbero essere usati per la funzione hash.

Se si ha la necessità di trattare lo stesso tipo in modi diversi per Dizionario / HashTables, è possibile utilizzare IEqualityComparer.

Override Equals e GetHashCode su tipi personalizzati

Per una classe `Person` come:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); //false because it's reference Equals
```

Ma definendo `Equals` e `GetHashCode` come segue:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }

    public override bool Equals(object obj)
```

```
    {
        var person = obj as Person;
        if(person == null) return false;
        return Name == person.Name && Age == person.Age; //the clothes are not important when
comparing two persons
    }

    public override int GetHashCode()
    {
        return Name.GetHashCode()*Age;
    }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); // result is true
```

Anche usando LINQ per fare query diverse su persone controllerà sia `Equals` che `GetHashCode` :

```
var persons = new List<Person>
{
     new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
     new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
     new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();//distinctPersons has Count = 2
```

Uguale a GetHashCode in IEqualityComparator

Per tipo di dato `Person` :

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

List<Person> persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
    new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();// distinctPersons has Count = 3
```

Ma definendo `Equals` e `GetHashCode` in `IEqualityComparator` :

```
public class PersonComparator : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name && x.Age == y.Age; //the clothes are not important when
```

```
comparing two persons;
    }

    public int GetHashCode(Person obj) { return obj.Name.GetHashCode() * obj.Age; }
}

var distinctPersons = persons.Distinct(new PersonComparator()).ToList();// distinctPersons has
Count = 2
```

Si noti che per questa query, due oggetti sono stati considerati uguali se entrambi gli `Equals` restituito true e il `GetHashCode` ha restituito lo stesso codice hash per le due persone.

Leggi Uguale e GetHashCode online: https://riptutorial.com/it/csharp/topic/3429/uguale-e-gethashcode

Capitolo 158: Una panoramica delle collezioni c

Examples

HashSet

Questa è una raccolta di oggetti unici, con ricerca O (1).

```
HashSet<int> validStoryPointValues = new HashSet<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(1)
```

A titolo di confronto, fare un `Contains` su un elenco produce prestazioni peggiori:

```
List<int> validStoryPointValues = new List<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(n)
```

`HashSet.Contains` utilizza una tabella hash, in modo che le ricerche siano estremamente veloci, indipendentemente dal numero di elementi nella raccolta.

SortedSet

```
// create an empty set
var mySet = new SortedSet<int>();

// add something
// note that we add 2 before we add 1
mySet.Add(2);
mySet.Add(1);

// enumerate through the set
foreach(var item in mySet)
{
    Console.WriteLine(item);
}

// output:
// 1
// 2
```

T [] (Matrice di T)

```
// create an array with 2 elements
var myArray = new [] { "one", "two" };

// enumerate through the array
foreach(var item in myArray)
{
    Console.WriteLine(item);
```

```
}

// output:
// one
// two

// exchange the element on the first position
// note that all collections start with the index 0
myArray[0] = "something else";


// enumerate through the array again
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
// something else
// two
```

Elenco

`List<T>` è un elenco di un determinato tipo. Gli articoli possono essere aggiunti, inseriti, rimossi e indirizzati per indice.

```
using System.Collections.Generic;

var list = new List<int>() { 1, 2, 3, 4, 5 };
list.Add(6);
Console.WriteLine(list.Count); // 6
list.RemoveAt(3);
Console.WriteLine(list.Count); // 5
Console.WriteLine(list[3]);    // 5
```

`List<T>` può essere pensato come una matrice che è possibile ridimensionare. L'enumerazione della raccolta in ordine è rapida, così come l'accesso ai singoli elementi tramite il loro indice. Per accedere a elementi basati su alcuni aspetti del loro valore, o qualche altra chiave, un `Dictionary<T>` fornirà una ricerca più rapida.

Dizionario

Dizionario <TKey, TValue> è una mappa. Per una determinata chiave ci può essere un valore nel dizionario.

```
using System.Collections.Generic;

var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}
};

// Reading data
Console.WriteLine(people["John"]); // 30
Console.WriteLine(people["George"]); // throws KeyNotFoundException
```

```
int age;
if (people.TryGetValue("Mary", out age))
{
    Console.WriteLine(age); // 35
}

// Adding and changing data
people["John"] = 40;    // Overwriting values this way is ok
people.Add("John", 40); // Throws ArgumentException since "John" already exists

// Iterating through contents
foreach(KeyValuePair<string, int> person in people)
{
    Console.WriteLine("Name={0}, Age={1}", person.Key, person.Value);
}

foreach(string name in people.Keys)
{
    Console.WriteLine("Name={0}", name);
}

foreach(int age in people.Values)
{
    Console.WriteLine("Age={0}", age);
}
```

Chiave duplicata quando si utilizza l'inizializzazione della raccolta

```
var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}, {"Jack", 40}
}; // throws ArgumentException since "Jack" already exists
```

Pila

```
// Initialize a stack object of integers
var stack = new Stack<int>();

// add some data
stack.Push(3);
stack.Push(5);
stack.Push(8);

// elements are stored with "first in, last out" order.
// stack from top to bottom is: 8, 5, 3

// We can use peek to see the top element of the stack.
Console.WriteLine(stack.Peek()); // prints 8

// Pop removes the top element of the stack and returns it.
Console.WriteLine(stack.Pop()); // prints 8
Console.WriteLine(stack.Pop()); // prints 5
```

```
Console.WriteLine(stack.Pop()); // prints 3
```

Lista collegata

```
// initialize a LinkedList of integers
LinkedList list = new LinkedList<int>();

// add some numbers to our list.
list.AddLast(3);
list.AddLast(5);
list.AddLast(8);

// the list currently is 3, 5, 8

list.AddFirst(2);
// the list now is 2, 3, 5, 8

list.RemoveFirst();
// the list is now 3, 5, 8

list.RemoveLast();
// the list is now 3, 5
```

Nota che `LinkedList<T>` rappresenta la lista *doppiamente* collegata. Quindi, è semplicemente la raccolta di nodi e ogni nodo contiene un elemento di tipo `T` Ogni nodo è collegato al nodo precedente e al nodo seguente.

Coda

```
// Initalize a new queue of integers
var queue = new Queue<int>();

// Add some data
queue.Enqueue(6);
queue.Enqueue(4);
queue.Enqueue(9);

// Elements in a queue are stored in "first in, first out" order.
// The queue from first to last is: 6, 4, 9

// View the next element in the queue, without removing it.
Console.WriteLine(queue.Peek()); // prints 6

// Removes the first element in the queue, and returns it.
Console.WriteLine(queue.Dequeue()); // prints 6
Console.WriteLine(queue.Dequeue()); // prints 4
Console.WriteLine(queue.Dequeue()); // prints 9
```

Filo sicuro! Utilizzare ConcurrentQueue in ambienti multi-thread.

Leggi Una panoramica delle collezioni c # online: https://riptutorial.com/it/csharp/topic/2344/una-panoramica-delle-collezioni-c-sharp

Capitolo 159: Uso della direttiva

Osservazioni

La parola chiave `using` è sia una direttiva (questo argomento) sia una dichiarazione.

Per l'istruzione `using` (vale a dire per incapsulare l'ambito di un oggetto `IDisposable` , assicurandosi che al di fuori di tale ambito l'oggetto venga eliminato in modo pulito), vedere Utilizzo di Statement .

Examples

Uso di base

```
using System;
using BasicStuff = System;
using Sayer = System.Console;
using static System.Console;  //From C# 6

class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Ignoring usings and specifying full type name");
        Console.WriteLine("Thanks to the 'using System' directive");
        BasicStuff.Console.WriteLine("Namespace aliasing");
        Sayer.WriteLine("Type aliasing");
        WriteLine("Thanks to the 'using static' directive (from C# 6)");
    }
}
```

Fai riferimento a uno spazio dei nomi

```
using System.Text;
//allows you to access classes within this namespace such as StringBuilder
//without prefixing them with the namespace.  i.e:

//...
var sb = new StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

Associare un alias con uno spazio dei nomi

```
using st = System.Text;
//allows you to access classes within this namespace such as StringBuilder
//prefixing them with only the defined alias and not the full namespace.  i.e:

//...
var sb = new st.StringBuilder();
```

```
//instead of
var sb = new System.Text.StringBuilder();
```

Accesso ai membri statici di una classe

6.0

Consente di importare un tipo specifico e utilizzare i membri statici del tipo senza qualificarli con il nome del tipo. Questo mostra un esempio usando metodi statici:

```
using static System.Console;

// ...

string GetName()
{
    WriteLine("Enter your name.");
    return ReadLine();
}
```

E questo mostra un esempio usando proprietà e metodi statici:

```
using static System.Math;

namespace Geometry
{
    public class Circle
    {
        public double Radius { get; set; };

        public double Area => PI * Pow(Radius, 2);
    }
}
```

Associare un alias per risolvere i conflitti

Se si utilizzano più spazi dei nomi che possono avere classi con lo stesso nome (come `System.Random` e `UnityEngine.Random`), è possibile utilizzare un alias per specificare che `Random` proviene da uno o l'altro senza dover utilizzare l'intero spazio dei nomi nella chiamata .

Per esempio:

```
using UnityEngine;
using System;

Random rnd = new Random();
```

Ciò farà sì che il compilatore non sia sicuro di quale `Random` valuterà la nuova variabile come. Invece, puoi fare:

```
using UnityEngine;
using System;
```

```
using Random = System.Random;

Random rnd = new Random();
```

Questo non ti preclude di chiamare l'altro dal suo spazio dei nomi completo, come questo:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
int unityRandom = UnityEngine.Random.Range(0,100);
```

`rnd` sarà una variabile `System.Random` e `unityRandom` sarà una variabile `UnityEngine.Random` .

Usando le direttive alias

È possibile utilizzare l' `using` per impostare un alias per uno spazio dei nomi o un tipo. Maggiori dettagli possono essere trovati qui .

Sintassi:

```
using <identifier> = <namespace-or-type-name>;
```

Esempio:

```
using NewType = Dictionary<string, Dictionary<string,int>>;
NewType multiDictionary = new NewType();
//Use instances as you are using the original one
multiDictionary.Add("test", new Dictionary<string,int>());
```

Leggi Uso della direttiva online: https://riptutorial.com/it/csharp/topic/52/uso-della-direttiva

Capitolo 160: Utilizzando json.net

introduzione

Utilizzo della classe JSONConverter di JSON.net .

Examples

Usando JsonConverter su valori semplici

Esempio utilizzando JsonCoverter per deserializzare la proprietà runtime dalla risposta api in un oggetto intervallo nel modello Film

JSON (http://www.omdbapi.com/?i=tt1663662)

```
{
    Title: "Pacific Rim",
    Year: "2013",
    Rated: "PG-13",
    Released: "12 Jul 2013",
    Runtime: "131 min",
    Genre: "Action, Adventure, Sci-Fi",
    Director: "Guillermo del Toro",
    Writer: "Travis Beacham (screenplay), Guillermo del Toro (screenplay), Travis Beacham
(story)",
    Actors: "Charlie Hunnam, Diego Klattenhoff, Idris Elba, Rinko Kikuchi",
    Plot: "As a war between humankind and monstrous sea creatures wages on, a former pilot and
a trainee are paired up to drive a seemingly obsolete special weapon in a desperate effort to
save the world from the apocalypse.",
    Language: "English, Japanese, Cantonese, Mandarin",
    Country: "USA",
    Awards: "Nominated for 1 BAFTA Film Award. Another 6 wins & 46 nominations.",
    Poster: "https://images-na.ssl-images-
amazon.com/images/M/MV5BMTY3MTI5NjQ4Nl5BMl5BanBnXkFtZTcwOTU1OTU0OQ@@._V1_SX300.jpg",
    Ratings: [{
            Source: "Internet Movie Database",
            Value: "7.0/10"
        },
        {
            Source: "Rotten Tomatoes",
            Value: "71%"
        },
        {
            Source: "Metacritic",
            Value: "64/100"
        }
    ],
    Metascore: "64",
    imdbRating: "7.0",
```

```
    imdbVotes: "398,198",
    imdbID: "tt1663662",
    Type: "movie",
    DVD: "15 Oct 2013",
    BoxOffice: "$101,785,482.00",
    Production: "Warner Bros. Pictures",
    Website: "http://pacificrimmovie.com",
    Response: "True"
}
```

Modello di film

```
using Project.Serializers;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.Threading.Tasks;

namespace Project.Models
{
    [DataContract]
    public class Movie
    {
        public Movie() { }

        [DataMember]
        public int Id { get; set; }

        [DataMember]
        public string ImdbId { get; set; }

        [DataMember]
        public string Title { get; set; }

        [DataMember]
        public DateTime Released { get; set; }

        [DataMember]
        [JsonConverter(typeof(RuntimeSerializer))]
        public TimeSpan Runtime { get; set; }

    }
}
```

RuntimeSerializer

```
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
```

```
using System.Threading.Tasks;

namespace Project.Serializers
{
    public class RuntimeSerializer : JsonConverter
    {
        public override bool CanConvert(Type objectType)
        {
            return objectType == typeof(TimeSpan);
        }

        public override object ReadJson(JsonReader reader, Type objectType, object
existingValue, JsonSerializer serializer)
        {
            if (reader.TokenType == JsonToken.Null)
                return null;

            JToken jt = JToken.Load(reader);
            String value = jt.Value<String>();

            Regex rx = new Regex("(\\s*)min$");
            value = rx.Replace(value, (m) => "");

            int timespanMin;
            if(!Int32.TryParse(value, out timespanMin))
            {
                throw new NotSupportedException();
            }

            return new TimeSpan(0, timespanMin, 0);
        }

        public override void WriteJson(JsonWriter writer, object value, JsonSerializer
serializer)
        {
            serializer.Serialize(writer, value);
        }
    }
}
```

Chiamandolo

```
Movie m = JsonConvert.DeserializeObject<Movie>(apiResponse));
```

Colleziona tutti i campi dell'oggetto JSON

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

public class JsonFieldsCollector
{
    private readonly Dictionary<string, JValue> fields;

    public JsonFieldsCollector(JToken token)
    {
        fields = new Dictionary<string, JValue>();
```

```
        CollectFields(token);
    }

    private void CollectFields(JToken jToken)
    {
        switch (jToken.Type)
        {
            case JTokenType.Object:
                foreach (var child in jToken.Children<JProperty>())
                    CollectFields(child);
                break;
            case JTokenType.Array:
                foreach (var child in jToken.Children())
                    CollectFields(child);
                break;
            case JTokenType.Property:
                CollectFields(((JProperty) jToken).Value);
                break;
            default:
                fields.Add(jToken.Path, (JValue)jToken);
                break;
        }
    }

    public IEnumerable<KeyValuePair<string, JValue>> GetAllFields() => fields;
}
```

Uso:

```
var json = JToken.Parse(/* JSON string */);
var fieldsCollector = new JsonFieldsCollector(json);
var fields = fieldsCollector.GetAllFields();

foreach (var field in fields)
    Console.WriteLine($"{field.Key}: '{field.Value}'");
```

dimostrazione
Per questo oggetto JSON

```
{
  "User": "John",
  "Workdays": {
    "Monday": true,
    "Tuesday": true,
    "Friday": false
  },
  "Age": 42
}
```

l'output atteso sarà:

```
User: 'John'
Workdays.Monday: 'True'
Workdays.Tuesday: 'True'
Workdays.Friday: 'False'
Age: '42'
```

Leggi Utilizzando json.net online: https://riptutorial.com/it/csharp/topic/9879/utilizzando-json-net

Capitolo 161: Utilizzando la dichiarazione

introduzione

Fornisce una comoda sintassi che garantisce l'uso corretto di oggetti IDisposable .

Sintassi

- usando (usa e getta) {}
- utilizzando (IDisposable disposable = new MyDisposable ()) {}

Osservazioni

L'oggetto `IDisposable` `using` deve implementare l'interfaccia `IDisposable` .

```
using(var obj = new MyObject())
{
}

class MyObject : IDisposable
{
    public void Dispose()
    {
        // Cleanup
    }
}
```

Esempi più completi per l'implementazione `IDisposable` possono essere trovati nei documenti MSDN .

Examples

Utilizzo delle nozioni di base sulle istruzioni

`using` dello zucchero sintattico consente di garantire che una risorsa venga pulita senza bisogno di un blocco `try-finally` esplicito. Questo significa che il tuo codice sarà molto più pulito e non perderai risorse non gestite.

Standard `Dispose` cleanup pattern, per gli oggetti che implementano l'interfaccia `IDisposable` (che la classe base `Stream` `FileStream` esegue in .NET):

```
int Foo()
{
    var fileName = "file.txt";

    {
        FileStream disposable = null;
```

```
        try
        {
            disposable = File.Open(fileName, FileMode.Open);

            return disposable.ReadByte();
        }
        finally
        {
            // finally blocks are always run
            if (disposable != null) disposable.Dispose();
        }
    }
}
```

`using` semplifica la sintassi nascondendo l'esplicito `try-finally` :

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        return disposable.ReadByte();
    }
    // disposable.Dispose is called even if we return earlier
}
```

Proprio come i blocchi `finally` eseguono sempre indipendentemente da errori o ritorni, `using` sempre chiamate `Dispose()` , anche in caso di errore:

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        throw new InvalidOperationException();
    }
    // disposable.Dispose is called even if we throw an exception earlier
}
```

Nota: Poiché si garantisce che `Dispose` sia chiamato indipendentemente dal flusso del codice, è una buona idea assicurarsi che `Dispose` non `Dispose` mai un'eccezione quando si implementa `IDisposable` . Altrimenti un'eccezione effettiva verrebbe sovrascritta dalla nuova eccezione che risulterebbe in un incubo di debug.

Ritornando dall'utilizzo del blocco

```
using ( var disposable = new DisposableItem() )
{
    return disposable.SomeProperty;
}
```

A causa delle semantica di `try..finally` cui il `using` blocco traduce, il `return` economico funziona

come previsto - il valore di ritorno viene valutata prima `finally` viene eseguito blocco e il valore disposto. L'ordine di valutazione è il seguente:

1. Valuta il corpo del `try`
2. Valuta e memorizza il valore restituito
3. Esegui infine il blocco
4. Restituisce il valore di ritorno memorizzato nella cache

Tuttavia, non è possibile restituire la variabile `disposable` e `disposable` stessa, poiché essa conterrebbe un riferimento disposto non valido - vedere l' esempio correlato .

Molteplici utilizzo di istruzioni con un blocco

È possibile utilizzare più nidificate `using` dichiarazioni senza aggiunta di più livelli di parentesi nidificate. Per esempio:

```
using (var input = File.OpenRead("input.txt"))
{
    using (var output = File.OpenWrite("output.txt"))
    {
        input.CopyTo(output);
    } // output is disposed here
} // input is disposed here
```

Un'alternativa è scrivere:

```
using (var input = File.OpenRead("input.txt"))
using (var output = File.OpenWrite("output.txt"))
{
    input.CopyTo(output);
} // output and then input are disposed here
```

Quale è esattamente equivalente al primo esempio.

Nota: le istruzioni nidificate `using` potrebbero attivare la regola CS2002 (vedere questa risposta per chiarimenti) e generare un avviso. Come spiegato nella risposta collegata, è generalmente sicuro annidare `using` istruzioni.

Quando i tipi all'interno dell'istruzione `using` sono dello stesso tipo, puoi separarli con virgole e specificare il tipo una sola volta (sebbene questo sia raro):

```
using (FileStream file = File.Open("MyFile.txt"), file2 = File.Open("MyFile2.txt"))
{
}
```

Questo può essere utilizzato anche quando i tipi hanno una gerarchia condivisa:

```
using (Stream file = File.Open("MyFile.txt"), data = new MemoryStream())
{
}
```

La parola chiave `var` *non può* essere utilizzata nell'esempio precedente. Si verificherebbe un errore di compilazione. Anche la dichiarazione separata da virgole non funzionerà quando le variabili dichiarate hanno tipi di gerarchie diverse.

Gotcha: restituire la risorsa che si sta smaltendo

La seguente è una cattiva idea perché eliminerebbe la variabile `db` prima di restituirla.

```
public IDBContext GetDBContext()
{
    using (var db = new DBContext())
    {
        return db;
    }
}
```

Questo può anche creare errori più sottili:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age);
    }
}
```

Questo sembra ok, ma il problema è che la valutazione delle espressioni LINQ è `DBContext` e probabilmente verrà eseguita solo in un secondo momento, quando il `DBContext` sottostante è già stato eliminato.

Quindi in breve l'espressione non viene valutata prima di lasciare l' `using` . Una possibile soluzione a questo problema, il che rende ancora uso di `using` , è di indurre l'espressione di valutare immediatamente chiamando un metodo che enumerare il risultato. Ad esempio `ToList()` , `ToArray()` , ecc. Se si utilizza la versione più recente di Entity Framework, è possibile utilizzare le controparti `async` come `ToListAsync()` o `ToArrayAsync()` .

Di seguito trovi l'esempio in azione:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age).ToList();
    }
}
```

È importante notare, tuttavia, che chiamando `ToList()` o `ToArray()` , l'espressione verrà valutata con entusiasmo, il che significa che tutte le persone con l'età specificata verranno caricate in memoria anche se non si esegue iterazione su di esse.

L'utilizzo delle istruzioni è sicuro

Non è necessario controllare l'oggetto `IDisposable` per `null` . `using` non genererà un'eccezione e `Dispose()` non verrà chiamato:

```
DisposableObject TryOpenFile()
{
    return null;
}

// disposable is null here, but this does not throw an exception
using (var disposable = TryOpenFile())
{
    // this will throw a NullReferenceException because disposable is null
    disposable.DoSomething();

    if(disposable != null)
    {
        // here we are safe because disposable has been checked for null
        disposable.DoSomething();
    }
}
```

Gotcha: Eccezione nel metodo Dispose che maschera altri errori in Uso dei blocchi

Considera il seguente blocco di codice.

```
try
{
    using (var disposable = new MyDisposable())
    {
        throw new Exception("Couldn't perform operation.");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

class MyDisposable : IDisposable
{
    public void Dispose()
    {
        throw new Exception("Couldn't dispose successfully.");
    }
}
```

Ci si può aspettare di vedere "Impossibile eseguire l'operazione" stampato sulla console ma si vedrebbe effettivamente "Impossibile eseguire correttamente l'operazione". poiché il metodo Dispose viene ancora chiamato anche dopo che la prima eccezione è stata lanciata.

Vale la pena essere consapevoli di questa sottigliezza poiché potrebbe nascondere l'errore reale che impediva la disposizione dell'oggetto e rendeva più difficile il debug.

Utilizzando le istruzioni e le connessioni al database

La parola chiave `using` assicura che la risorsa definita all'interno dell'istruzione esista solo nell'ambito dell'istruzione stessa. Qualsiasi risorsa definita all'interno dell'istruzione deve implementare l'interfaccia `IDisposable` .

Questi sono incredibilmente importanti quando si tratta di connessioni che implementano l'interfaccia `IDisposable` in quanto possono garantire che le connessioni non solo siano chiuse correttamente ma che le loro risorse siano liberate dopo che l'istruzione `using` è fuori dall'ambito.

Classi di dati comuni `IDisposable`

Molti dei seguenti sono classi correlate ai dati che implementano l'interfaccia `IDisposable` e sono candidati perfetti per una dichiarazione `using` :

- `SqlConnection` , `SqlCommand` , `SqlDataReader` , ecc.
- `OleDbConnection` , `OleDbCommand` , `OleDbDataReader` , ecc.
- `MySqlConnection` , `MySqlCommand` , `MySqlDbDataReader` , ecc.
- `DbContext`

Tutti questi sono comunemente usati per accedere ai dati tramite C # e verranno comunemente riscontrati durante la creazione di applicazioni incentrate sui dati. `FooDataReader` si può aspettare che molte altre classi che non sono menzionate che implementano le stesse `FooConnection` , `FooCommand` , `FooDataReader` si comportino allo stesso modo.

Modello di accesso comune per connessioni ADO.NET

Un modello comune che può essere utilizzato quando si accede ai dati tramite una connessione ADO.NET potrebbe essere il seguente:

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}")
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property");
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
         // Open your connection
         connection.Open();

         // Add your parameters here if necessary

         // Execute your query as a reader (again scoped with a using statement)
         using(var reader = command.ExecuteReader())
         {
               // Iterate through your results here
         }
    }
}
```

O se stavi semplicemente eseguendo un semplice aggiornamento e non avessi bisogno di un lettore, si applicherebbe lo stesso concetto di base:

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
     var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
     using(var command = new SqlCommand(query,connection))
     {
          connection.Open();

          // Add parameters here

          // Perform your update
          command.ExecuteNonQuery();
     }
}
```

Utilizzare le istruzioni con DataContexts

Molti ORM come Entity Framework espongono le classi di astrazione utilizzate per interagire con i database sottostanti sotto forma di classi come `DbContext` . Questi contesti generalmente implementano anche l'interfaccia `IDisposable` e dovrebbero trarne vantaggio attraverso l' `using` dichiarazioni quando possibile:

```
using(var context = new YourDbContext())
{
      // Access your context and perform your query
      var data = context.Widgets.ToList();
}
```

Utilizzare Dispose Syntax per definire l'ambito personalizzato

Per alcuni casi d'uso, è possibile utilizzare la sintassi `using` per aiutare a definire un ambito personalizzato. Ad esempio, è possibile definire la seguente classe per eseguire codice in una cultura specifica.

```
public class CultureContext : IDisposable
{
    private readonly CultureInfo originalCulture;

    public CultureContext(string culture)
    {
        originalCulture = CultureInfo.CurrentCulture;
        Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
    }

    public void Dispose()
    {
        Thread.CurrentThread.CurrentCulture = originalCulture;
    }
}
```

È quindi possibile utilizzare questa classe per definire blocchi di codice che vengono eseguiti in una cultura specifica.

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
```

```
using (new CultureContext("nl-NL"))
{
    // Code in this block uses the "nl-NL" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25-12-2016 00:00:00
}

using (new CultureContext("es-ES"))
{
    // Code in this block uses the "es-ES" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25/12/2016 0:00:00
}

// Reverted back to the original culture
Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 12/25/2016 12:00:00 AM
```

Nota: poiché non utilizziamo l'istanza di `CultureContext` che creiamo, non assegniamo una variabile per questo.

Questa tecnica viene utilizzata dal `BeginForm` aiutante in ASP.NET MVC.

Esecuzione del codice nel contesto del vincolo

Se si dispone di codice (una *routine*) che si desidera eseguire in un contesto specifico (vincolo), è possibile utilizzare l'iniezione della dipendenza.

L'esempio seguente mostra il vincolo di esecuzione in una connessione SSL aperta. Questa prima parte sarà nella libreria o nel framework, che non esporrai al codice cliente.

```
public static class SSLContext
{
    // define the delegate to inject
    public delegate void TunnelRoutine(BinaryReader sslReader, BinaryWriter sslWriter);

    // this allows the routine to be executed under SSL
    public static void ClientTunnel(TcpClient tcpClient, TunnelRoutine routine)
    {
        using (SslStream sslStream = new SslStream(tcpClient.GetStream(), true, _validate))
        {
            sslStream.AuthenticateAsClient(HOSTNAME, null, SslProtocols.Tls, false);

            if (!sslStream.IsAuthenticated)
            {
                throw new SecurityException("SSL tunnel not authenticated");
            }

            if (!sslStream.IsEncrypted)
            {
                throw new SecurityException("SSL tunnel not encrypted");
            }

            using (BinaryReader sslReader = new BinaryReader(sslStream))
            using (BinaryWriter sslWriter = new BinaryWriter(sslStream))
            {
                routine(sslReader, sslWriter);
            }
        }
```

```
    }
}
```

Ora il codice client che vuole fare qualcosa sotto SSL ma non vuole gestire tutti i dettagli SSL. Ora puoi fare tutto ciò che vuoi all'interno del tunnel SSL, ad esempio scambiare una chiave simmetrica:

```
public void ExchangeSymmetricKey(BinaryReader sslReader, BinaryWriter sslWriter)
{
    byte[] bytes = new byte[8];
    (new RNGCryptoServiceProvider()).GetNonZeroBytes(bytes);
    sslWriter.Write(BitConverter.ToUInt64(bytes, 0));
}
```

Esegui questa routine come segue:

```
SSLContext.ClientTunnel(tcpClient, this.ExchangeSymmetricKey);
```

Per fare ciò, è necessaria la clausola `using()` perché è l'unico modo (a parte un `try..finally` block) che puoi garantire che il codice client (`ExchangeSymmetricKey`) non esca mai senza disporre correttamente delle risorse usa e getta. Senza la clausola `using()` , non si potrebbe mai sapere se una routine potrebbe rompere il vincolo del contesto per smaltire tali risorse.

Leggi Utilizzando la dichiarazione online: https://riptutorial.com/it/csharp/topic/38/utilizzando-la-dichiarazione

Capitolo 162: Utilizzo di SQLite in C

Examples

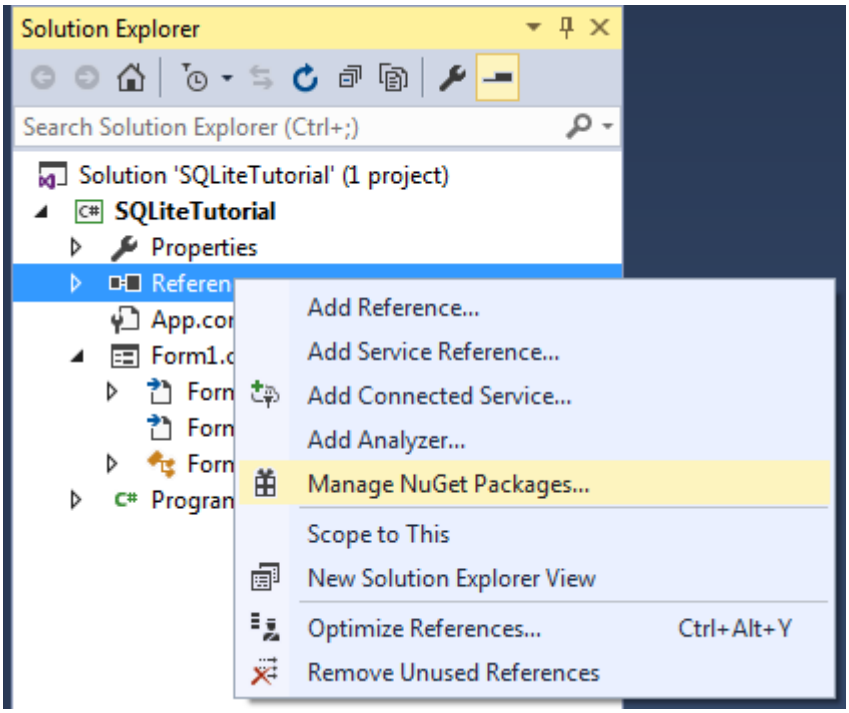
Creazione di CRUD semplice con SQLite in C

Prima di tutto è necessario aggiungere il supporto SQLite alla nostra applicazione. Ci sono due modi per farlo

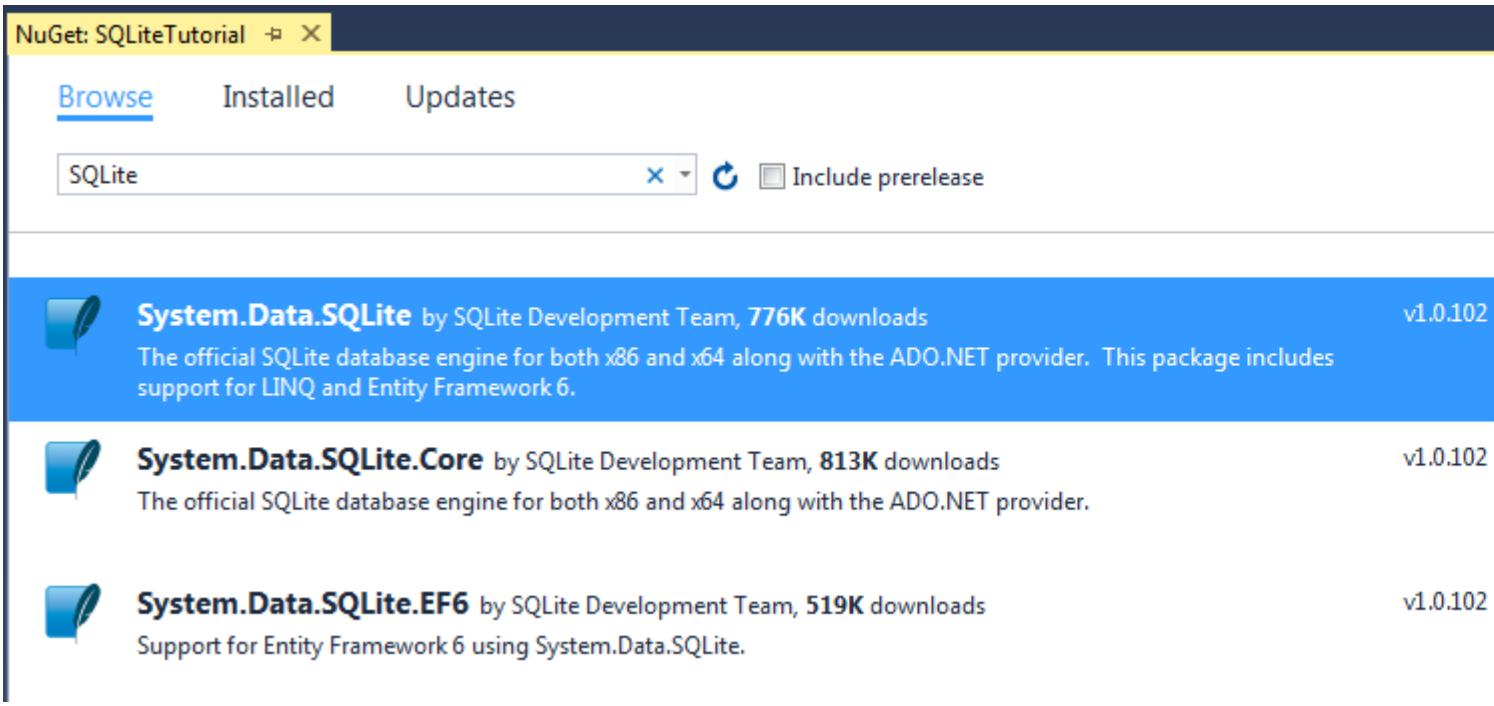
- Scarica la DLL che soddisfa il tuo sistema dalla pagina di download di SQLite e quindi aggiungi manualmente al progetto
- Aggiungi dipendenza SQLite tramite NuGet

Lo faremo nel secondo modo

Per prima cosa apri il menu NuGet



e cercare **System.Data.SQLite** , selezionarlo e premere **Installa**



L'installazione può essere eseguita anche dalla console di Gestione pacchetti con

```
PM> Install-Package System.Data.SQLite
```

O solo per le funzionalità principali

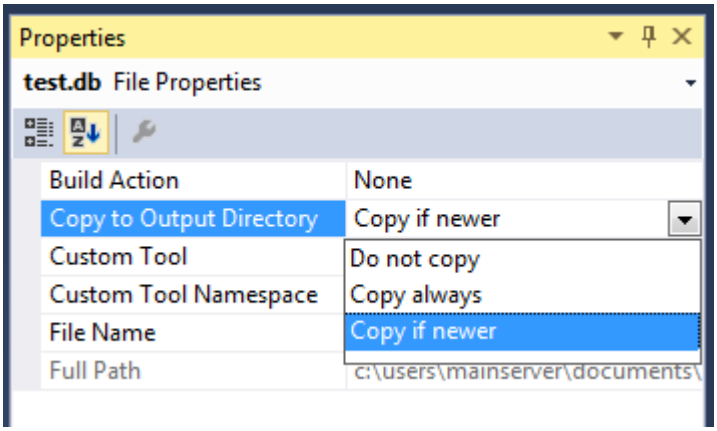
```
PM> Install-Package System.Data.SQLite.Core
```

Questo è tutto per il download, quindi possiamo andare direttamente nella programmazione.

Prima crea un semplice database SQLite con questa tabella e aggiungilo come un file al progetto

```
CREATE TABLE User(
  Id INTEGER PRIMARY KEY AUTOINCREMENT,
  FirstName TEXT NOT NULL,
  LastName TEXT NOT NULL
);
```

Inoltre, non dimenticare di impostare la proprietà **Copia su Output Directory** del file su **Copia se più recente** di **Copia sempre** , in base alle proprie esigenze



Crea una classe chiamata User, che sarà l'entità di base per il nostro database

```
private class User
{
    public string FirstName { get; set; }
    public string Lastname { get; set; }
}
```

Scriveremo due metodi per l'esecuzione della query, il primo per l'inserimento, l'aggiornamento o la rimozione dal database

```
private int ExecuteWrite(string query, Dictionary<string, object> args)
{
    int numberOfRowsAffected;

    //setup the connection to the database
    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();

        //open a new command
        using (var cmd = new SQLiteCommand(query, con))
        {
            //set the arguments given in the query
            foreach (var pair in args)
            {
                cmd.Parameters.AddWithValue(pair.Key, pair.Value);
            }

            //execute the query and get the number of row affected
            numberOfRowsAffected = cmd.ExecuteNonQuery();
        }

        return numberOfRowsAffected;
    }
}
```

e il secondo per la lettura dal database

```
private DataTable Execute(string query)
{
    if (string.IsNullOrEmpty(query.Trim()))
        return null;

    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();
        using (var cmd = new SQLiteCommand(query, con))
        {
            foreach (KeyValuePair<string, object> entry in args)
            {
                cmd.Parameters.AddWithValue(entry.Key, entry.Value);
            }

            var da = new SQLiteDataAdapter(cmd);

            var dt = new DataTable();
            da.Fill(dt);
```

```
            da.Dispose();
            return dt;
        }
    }
}
```

Ora passiamo ai nostri metodi **CRUD**

Aggiungere utente

```
private int AddUser(User user)
{
    const string query = "INSERT INTO User(FirstName, LastName) VALUES(@firstName,
@lastName)";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@firstName", user.FirstName},
        {"@lastName", user.Lastname}
    };

    return ExecuteWrite(query, args);
}
```

Modifica utente

```
private int EditUser(User user)
{
    const string query = "UPDATE User SET FirstName = @firstName, LastName = @lastName WHERE
Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id},
        {"@firstName", user.FirstName},
        {"@lastName", user.Lastname}
    };

    return ExecuteWrite(query, args);
}
```

Eliminazione dell'utente

```
private int DeleteUser(User user)
{
    const string query = "Delete from User WHERE Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id}
```

```
    };

    return ExecuteWrite(query, args);
}
```

Ottenere l'utente da `Id`

```
private User GetUserById(int id)
{
    var query = "SELECT * FROM User WHERE Id = @id";

    var args = new Dictionary<string, object>
    {
        {"@id", id}
    };

    DataTable dt = ExecuteRead(query, args);

    if (dt == null || dt.Rows.Count == 0)
    {
        return null;
    }

    var user = new User
    {
        Id = Convert.ToInt32(dt.Rows[0]["Id"]),
        FirstName = Convert.ToString(dt.Rows[0]["FirstName"]),
        Lastname = Convert.ToString(dt.Rows[0]["LastName"])
    };

    return user;
}
```

Esecuzione della query

```
using (SQLiteConnection conn = new SQLiteConnection(@"Data
Source=data.db;Pooling=true;FailIfMissing=false"))
{
    conn.Open();
    using (SQLiteCommand cmd = new SQLiteCommand(conn))
    {
       cmd.CommandText = "query";
       using (SqlDataReader dr = cmd.ExecuteReader())
       {
           while(dr.Read())
           {
               //do stuff
           }
       }
    }
}
```

Nota : l'impostazione `FailIfMissing` su true crea il file `data.db` se mancante. Tuttavia, il file sarà vuoto. Quindi, qualsiasi tabella richiesta deve essere ricreata.

Leggi Utilizzo di SQLite in C # online: https://riptutorial.com/it/csharp/topic/4960/utilizzo-di-sqlite-in-

c-sharp

Capitolo 163: Windows Communication Foundation

introduzione

Windows Communication Foundation (WCF) è un framework per la creazione di applicazioni orientate ai servizi. Utilizzando WCF, è possibile inviare dati come messaggi asincroni da un endpoint di un servizio a un altro. Un endpoint del servizio può essere parte di un servizio continuamente disponibile ospitato da IIS o può essere un servizio ospitato in un'applicazione. I messaggi possono essere semplici come un singolo carattere o una parola inviata come XML o complessa come un flusso di dati binari.

Examples

Iniziare campione

Il servizio descrive le operazioni che esegue in un contratto di servizio che espone pubblicamente come metadati.

```
// Define a service contract.
[ServiceContract(Namespace="http://StackOverflow.ServiceModel.Samples")]
public interface ICalculator
{
    [OperationContract]
    double Add(double n1, double n2);
}
```

L'implementazione del servizio calcola e restituisce il risultato appropriato, come mostrato nel seguente codice di esempio.

```
// Service class that implements the service contract.
public class CalculatorService : ICalculator
{
    public double Add(double n1, double n2)
    {
        return n1 + n2;
    }
}
```

Il servizio espone un endpoint per la comunicazione con il servizio, definito utilizzando un file di configurazione (Web.config), come mostrato nella seguente configurazione di esempio.

```
<services>
    <service
        name="StackOverflow.ServiceModel.Samples.CalculatorService"
        behaviorConfiguration="CalculatorServiceBehavior">
        <!-- ICalculator is exposed at the base address provided by
```

```
         host: http://localhost/servicemodelsamples/service.svc.  -->
       <endpoint address=""
             binding="wsHttpBinding"
             contract="StackOverflow.ServiceModel.Samples.ICalculator" />
       ...
    </service>
</services>
```

Il framework non espone i metadati per impostazione predefinita. In quanto tale, il servizio attiva ServiceMetadataBehavior ed espone un endpoint MEX (metadata exchange) su http://localhost/servicemodelsamples/service.svc/mex . La seguente configurazione lo dimostra.

```
<system.serviceModel>
  <services>
    <service
        name="StackOverflow.ServiceModel.Samples.CalculatorService"
        behaviorConfiguration="CalculatorServiceBehavior">
      ...
      <!-- the mex endpoint is explosed at
       http://localhost/servicemodelsamples/service.svc/mex -->
      <endpoint address="mex"
                binding="mexHttpBinding"
                contract="IMetadataExchange" />
    </service>
  </services>

  <!--For debugging purposes set the includeExceptionDetailInFaults
   attribute to true-->
  <behaviors>
    <serviceBehaviors>
      <behavior name="CalculatorServiceBehavior">
        <serviceMetadata httpGetEnabled="True"/>
        <serviceDebug includeExceptionDetailInFaults="False" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

Il client comunica utilizzando un determinato tipo di contratto utilizzando una classe client generata da ServiceModel Metadata Utility Tool (Svcutil.exe).

Eseguire il seguente comando dal prompt dei comandi SDK nella directory del client per generare il proxy digitato:

```
svcutil.exe /n:"http://StackOverflow.ServiceModel.Samples,StackOverflow.ServiceModel.Samples"
http://localhost/servicemodelsamples/service.svc/mex /out:generatedClient.cs
```

Come il servizio, il client utilizza un file di configurazione (App.config) per specificare l'endpoint con il quale desidera comunicare. La configurazione dell'endpoint del client è costituita da un indirizzo assoluto per l'endpoint del servizio, l'associazione e il contratto, come illustrato nell'esempio seguente.

```
<client>
     <endpoint
         address="http://localhost/servicemodelsamples/service.svc"
```

```
        binding="wsHttpBinding"
        contract="StackOverflow.ServiceModel.Samples.ICalculator" />
</client>
```

L'implementazione client crea un'istanza del client e utilizza l'interfaccia tipizzata per iniziare a comunicare con il servizio, come mostrato nel seguente codice di esempio.

```
// Create a client.
CalculatorClient client = new CalculatorClient();

// Call the Add service operation.
double value1 = 100.00D;
double value2 = 15.99D;
double result = client.Add(value1, value2);
Console.WriteLine("Add({0},{1}) = {2}", value1, value2, result);

//Closing the client releases all communication resources.
client.Close();
```

Leggi Windows Communication Foundation online:
https://riptutorial.com/it/csharp/topic/10760/windows-communication-foundation

Capitolo 164: XDocument e lo spazio dei nomi System.Xml.Linq

Examples

Genera un documento XML

L'obiettivo è generare il seguente documento XML:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit ID="F0001">
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit ID="F0002">
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Codice:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDeclaration xDeclaration = new XDeclaration("1.0", "utf-8", "yes");
XDocument xDoc = new XDocument(xDeclaration);
XElement xRoot = new XElement(xns + "FruitBasket");
xDoc.Add(xRoot);

XElement xelFruit1 = new XElement(xns + "Fruit");
XAttribute idAttribute1 = new XAttribute("ID", "F0001");
xelFruit1.Add(idAttribute1);
XElement xelFruitName1 = new XElement(xns + "FruitName", "Banana");
XElement xelFruitColor1 = new XElement(xns + "FruitColor", "Yellow");
xelFruit1.Add(xelFruitName1);
xelFruit1.Add(xelFruitColor1);
xRoot.Add(xelFruit1);

XElement xelFruit2 = new XElement(xns + "Fruit");
XAttribute idAttribute2 = new XAttribute("ID", "F0002");
xelFruit2.Add(idAttribute2);
XElement xelFruitName2 = new XElement(xns + "FruitName", "Apple");
XElement xelFruitColor2 = new XElement(xns + "FruitColor", "Red");
xelFruit2.Add(xelFruitName2);
xelFruit2.Add(xelFruitColor2);
xRoot.Add(xelFruit2);
```

Modifica file XML

Per modificare un file XML con `XDocument` , devi caricare il file in una variabile di tipo `XDocument` , modificarlo in memoria, quindi salvarlo, sovrascrivendo il file originale. Un errore comune consiste nel modificare l'XML in memoria e aspettarsi che il file sul disco cambi.

Dato un file XML:

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Vuoi modificare il colore della banana in marrone:

1. Dobbiamo conoscere il percorso del file su disco.
2. Un sovraccarico di `XDocument.Load` riceve un URI (percorso del file).
3. Poiché il file xml utilizza uno spazio dei nomi, è necessario eseguire una query con lo spazio dei nomi E il nome dell'elemento.
4. Una query Linq che utilizza la sintassi C # 6 per tenere conto della possibilità di valori nulli. Ogni `.` usato in questa query ha il potenziale per restituire un set nullo se la condizione non trova elementi. Prima del C # 6 lo farebbe in più passaggi, controllando la nullità lungo il percorso. Il risultato è l'elemento `<Fruit>` che contiene la banana. In realtà un `IEnumerable<XElement>` , che è il motivo per cui il prossimo passo utilizza `FirstOfDefault()` .
5. Ora estraiamo l'elemento FruitColor dall'elemento Fruit appena trovato. Qui assumiamo che ce ne sia solo uno, o ci interessa solo il primo.
6. Se non è nullo, impostiamo il FruitColor su "Brown".
7. Infine, salviamo l' `XDocument` , sovrascrivendo il file originale su disco.

```
// 1.
string xmlFilePath = "c:\\users\\public\\fruit.xml";

// 2.
XDocument xdoc = XDocument.Load(xmlFilePath);

// 3.
XNamespace ns = "http://www.fruitauthority.fake";

//4.
var elBanana = xdoc.Descendants()?.
    Elements(ns + "FruitName")?.
    Where(x => x.Value == "Banana")?.
    Ancestors(ns + "Fruit");

// 5.
var elColor = elBanana.Elements(ns + "FruitColor").FirstOrDefault();

// 6.
if (elColor != null)
{
    elColor.Value = "Brown";
}

// 7.
```

```
xdoc.Save(xmlFilePath);
```

Il file ora si presenta così:

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Brown</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Genera un documento XML usando la sintassi fluente

Obbiettivo:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Codice:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDocument xDoc =
    new XDocument(new XDeclaration("1.0", "utf-8", "yes"),
        new XElement(xns + "FruitBasket",
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Banana"),
                new XElement(xns + "FruitColor", "Yellow")),
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Apple"),
                new XElement(xns + "FruitColor", "Red"))
                ));
```

Leggi XDocument e lo spazio dei nomi System.Xml.Linq online:
https://riptutorial.com/it/csharp/topic/1866/xdocument-e-lo-spazio-dei-nomi-system-xml-linq

Capitolo 165: XmlDocument e lo spazio dei nomi System.Xml

Examples

Interazione documento XML di base

```
public static void Main()
{
    var xml  = new XmlDocument();
    var root = xml.CreateElement("element");
        // Creates an attribute, so the element will now be "<element attribute='value' />"
        root.SetAttribute("attribute", "value");

    // All XML documents must have one, and only one, root element
    xml.AppendChild(root);

    // Adding data to an XML document
    foreach (var dayOfWeek in Enum.GetNames((typeof(DayOfWeek))))
    {
        var day = xml.CreateElement("dayOfWeek");
            day.SetAttribute("name", dayOfWeek);

        // Don't forget to add the new value to the current document!
        root.AppendChild(day);
    }

    // Looking for data using XPath; BEWARE, this is case-sensitive
    var monday = xml.SelectSingleNode("//dayOfWeek[@name='Monday']");
    if (monday != null)
    {
        // Once you got a reference to a particular node, you can delete it
        // by navigating through its parent node and asking for removal
        monday.ParentNode.RemoveChild(monday);
    }

    // Displays the XML document in the screen; optionally can be saved to a file
    xml.Save(Console.Out);
}
```

Lettura dal documento XML

Un esempio di file XML

```
    <Sample>
    <Account>
        <One number="12"/>
        <Two number="14"/>
    </Account>
    <Account>
        <One number="14"/>
        <Two number="16"/>
    </Account>
```

```
    </Sample>
```

Leggendo da questo file XML:

```
    using System.Xml;
    using System.Collections.Generic;

    public static void Main(string fullpath)
    {
        var xmldoc = new XmlDocument();
        xmldoc.Load(fullpath);

        var oneValues = new List<string>();

        // Getting all XML nodes with the tag name
        var accountNodes = xmldoc.GetElementsByTagName("Account");
        for (var i = 0; i < accountNodes.Count; i++)
        {
            // Use Xpath to find a node
            var account = accountNodes[i].SelectSingleNode("./One");
            if (account != null && account.Attributes != null)
            {
                // Read node attribute
                oneValues.Add(account.Attributes["number"].Value);
            }
        }
}
```

XmlDocument vs XDocument (esempio e confronto)

Esistono diversi modi per interagire con un file Xml.

1. Documento Xml
2. XDocument
3. XmlReader / XmlWriter

Prima di LINQ to XML eravamo usati XMLDocument per le manipolazioni in XML come l'aggiunta di attributi, elementi e così via. Ora LINQ to XML utilizza XDocument per lo stesso tipo di cose. Le sintassi sono molto più semplici di XMLDocument e richiedono una quantità minima di codice.

Anche XDocument è mutch più veloce come XmlDocument. XmlDoucument è una soluzione vecchia e sporca per eseguire query su un documento XML.

Ho intenzione di mostrare alcuni esempi di classe XmlDocument e classe di classe **XDocument** :

Carica il file XML

```
string filename = @"C:\temp\test.xml";
```

XmlDocument

```
XmlDocument _doc = new XmlDocument();
_doc.Load(filename);
```

XDocument

```
XDocument _doc = XDocument.Load(fileName);
```

Crea XmlDocument

XmlDocument

```
XmlDocument doc = new XmlDocument();
XmlElement root = doc.CreateElement("root");
root.SetAttribute("name", "value");
XmlElement child = doc.CreateElement("child");
child.InnerText = "text node";
root.AppendChild(child);
doc.AppendChild(root);
```

XDocument

```
 XDocument doc = new XDocument(
    new XElement("Root", new XAttribute("name", "value"),
    new XElement("Child", "text node"))
);

/*result*/
<root name="value">
    <child>"TextNode"</child>
</root>
```

Cambia InnerText del nodo in XML

XmlDocument

```
XmlNode node = _doc.SelectSingleNode("xmlRootNode");
node.InnerText = value;
```

XDocument

```
 XElement rootNote = _doc.XPathSelectElement("xmlRootNode");
rootNode.Value = "New Value";
```

Salva il file dopo la modifica

Assicurati di proteggere l'xml dopo ogni modifica.

```
// Safe XmlDocument and XDocument
_doc.Save(filename);
```

Recupera i valori da XML

XmlDocument

```
 XmlNode node = _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
string text = node.InnerText;
```

XDocument

```
 XElement node = _doc.XPathSelectElement("xmlRootNode/levelOneChildNode");
 string text = node.Value;
```

Recupera il valore da tutti da tutti gli elementi figlio dove attributo = qualcosa.

XmlDocument

```
List<string> valueList = new List<string>();
    foreach (XmlNode n in nodelist)
    {
        if(n.Attributes["type"].InnerText == "City")
        {
            valueList.Add(n.Attributes["type"].InnerText);
        }
    }
```

XDocument

```
var accounts = _doc.XPathSelectElements("/data/summary/account").Where(c =>
c.Attribute("type").Value == "setting").Select(c => c.Value);
```

Aggiungi un nodo

XmlDocument

```
XmlNode nodeToAppend = doc.CreateElement("SecondLevelNode");
nodeToAppend.InnerText = "This title is created by code";

/* Append node to parent */
XmlNode firstNode= _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
firstNode.AppendChild(nodeToAppend);

/*After a change make sure to safe the document*/
_doc.Save(fileName);
```

XDocument

```
_doc.XPathSelectElement("ServerManagerSettings/TcpSocket").Add(new
XElement("SecondLevelNode"));

 /*After a change make sure to safe the document*/
_doc.Save(fileName);
```

Leggi XmlDocument e lo spazio dei nomi System.Xml online:
https://riptutorial.com/it/csharp/topic/1528/xmldocument-e-lo-spazio-dei-nomi-system-xml

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con C # Language	4444, A. Raza, A_Arnold , aalaap, Aaron Hudon, abishekshivan, Ade Stringer, Aleksandur Murfitt, Almir Vuk, Alok Singh, Andrii Abramov, AndroidMechanic, Aravind Suresh, Artemix, Ben Aaronson, Bernard Vander Beken, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, bpoiss, Br0k3nL1m1ts, Callum Watkins, Carlos Muñoz, Chad Levy, Chris Nantau, Christopher Ronning, Community, Confiqure, crunchy, David G., David Pine, DavidG, DAXaholic , Delphi.Boy, Durgpal Singh, DWright, Ehsan Sajjad, Elie Saad, Emre Bolat, enrico.bacis, fabriciorissetto, FadedAce, Florian Greinacher, Florian Koch , Frankenstine Joe, Gennady Trubach, GingerHead, Gordon Bell, gracacs, G-Wiz, H. Pauwelyn, Happypig375, Henrik H, HodofHod, Hywel Rees, iliketocode, Iordanis , Jamie Rees, Jawa, jnovo, John Slegers, Kayathiri, ken2k , Kevin Montrose, Kritner , Krzyserious, leumas1960, M Monis

		Ahmed Khan, Mahmoud Elgindy, Malick, Marcus Höglund, Mateen Ulhaq, Matt, Matt, Matt, Matt, Michael B, Michael Brandon Morris, Miljen Mikic, Millan Sanchez, Nate Barbettini, Nick, Nick Cox, Nipun Tripathi, NotMyself, Ojen, PashaPash, pijemcolu, Prateek, Raj Rao, Rajput , Rakitić, Rion Williams, RokumDev, RomCoo, Ryan Hilbert, sebingel, SeeuD1, solidcell, Steven Ackley, sumit sharma, Tofix, Tom Bowers, Travis J, Tushar patel, User 00000, user3185569, Ven, Victor Tomaili, viggity, void, Wen Qin, Ziad Akiki , Zze
2	.NET Compiler Platform (Roslyn)	4444, Lukáš Lánský
3	Accedi alla cartella condivisa di rete con nome utente e password	Mohsin khan
4	Accesso ai database	ATechieThought, ravindra, Rion Williams, The_Outsider, user2321864
5	Alberi di espressione	Benjamin Hodgson, dasblinkenlight, Dileep, George Duckett, just.another.programmer , Matas Vaitkevicius, matteeyah, meJustAndrew, Nathan Tuggy, NikolayKondratyev, Rob, Ruben Steins, Stephen Leppik, Раҳул Мақвана
6	Alias di tipi predefiniti	Racil Hilan, Rahul Nikate

		, Stephen Leppik
7	Annotazione dei dati	Maxime, Mikko Viitala, The_Outsider, Will Ray
8	Argomenti nominati	Cihan Yakar, Danny Chen, mehrandvd, Pan, Pavel Mayorov, Stephen Leppik
9	Argomenti nominati e opzionali	RamenChef, Sibeesh Venu, Testing123, The_Outsider, Tim Yusupov
10	Array	A_Arnold, Aaron Hudon, Alexey Groshev, Anas Tasadduq, Andrii Abramov, Baddie, Benjamin Hodgson, bluray, coyote, D.J., das_keyboard, Fernando Matsumoto, granmirupa, Jaydip Jadhav, Jeppe Stig Nielsen, Jon Schneider, Ogoun, RamenChef, Robert Columbia, Shyju, The_Outsider, Thomas Weller, tonirush, Tormod Haugene, Wasabi Fan, Wen Qin, Xiobiq, Yotam Salmon
11	AssemblyInfo.cs Esempi	Adi Lester, Ameya Deshpande, AndreyAkinshin, Boggin, Dodzi Dzakuma, dove, Joel Martinez, pinkfloydx33, Ralf Bönning, Theodoros Chatzigiannakis, Wasabi Fan
12	Async-Await	Aaron Hudon, AGB, aholmes, Ant P, Benjol, BrunoLM, Conrad.Dean, Craig Brett, Donald

		Webb, EJoshuaS, EvilTak, gdyrrahitis, George Duckett, Grimm The Opiner, Guanxi, guntbert, H. Pauwelyn, jdpilgrim, ken2k, Kevin Montrose, marshal craft, Michael Richardson, Moerwald, Nate Barbettini, nickguletskii, Pavel Mayorov, Pavel Voronin, pinkfloydx33, Rob, Serg Rogovtsev, Stefano d'Antonio, Stephen Leppik, SynerCoder, trashr0x, Tseng, user2321864, Vincent
13	attributi	Alexander Mandt, Andrew Diamond, Doruk, LosManos, Lukas Kolletzki, NikolayKondratyev, Pavel Sapehin, SysVoid, TKharaishvili
14	BackgroundWorker	Bovaz, Draken, ephtee, Jacobr365, Will
15	BigInteger	4444, Ed Marty, James Hughes, Rob, The_Outsider
16	BindingList	Bovaz, Stephen Leppik, yumaikas
17	C # 3.0 Caratteristiche	0xFF, bob0the0mighty, FrenkyB, H. Pauwelyn, ken2k, Maniero, Rob
18	C # 4.0 Caratteristiche	Benjamin Hodgson, Botond Balázs, H. Pauwelyn, Proxima, Sibeesh Venu, Squidward, Theodoros Chatzigiannakis

19	C # 5.0 Caratteristiche	Abob, alex.b, H. Pauwelyn
20	C # 6.0 Caratteristiche	A_Arnold, Aaron Anodide, Aaron Hudon, Adil Mammadov, Adriano Repetti, AER, AGB, Akshay Anand, Alan McBee, Alex Logan, Amitay Stern, anaximander, andre_ss6 , Andrea, AndroidMechanic, Ares, Arthur Rizzo, Ashwin Ramaswami, avishayp, Balagurunathan Marimuthu, Bardia, Ben Aaronson, Blubberguy22 , Bobson, bpoiss, Bradley Uffner, Bret Copeland, C4u, Callum Watkins, Chad Levy, Charlie H, ChrFin, Community, Conrad.Dean, Cyprien Autexier, Dan, Daniel Minnaar, Daniel Stradowski, DarkV1, dasblinkenlight, David, David G., David Pine, Deepak gupta, DLeh, dotctor, Durgpal Singh, Ehsan Sajjad, el2iot2, Emre Bolat, enrico.bacis, Erik Schierboom, fabriciorissetto, faso, Franck Dernoncourt, FrankerZ, Gabor Kecskemeti, Gary, Gates Wong, Geoff, GingerHead, Gordon Bell, Guillaume Pascal, H. Pauwelyn, hankide, Henrik H, iliketocode, Iordanis , Irfan, Ivan Yurchenko, J. Steen, Jacob Linney, Jamie

Rees, Jason Sturges, Jeppe Stig Nielsen, Jim, JNYRanger, Joe, Joel Etherton, John Slegers, Johnbot, Jojodmo, Jonas S, Juan, Kapep, ken2k, Kit, Konamiman, Krikor Ailanjian, Lafexlos, LaoR , Lasse Vågsæther Karlsen, M.kazem Akhgary, Mafii, Magisch, Makyen, MANISH KUMAR CHOUDHARY, Marc, MarcinJuraszek, Mark Shevchenko, Matas Vaitkevicius, Mateen Ulhaq, Matt, Matt, Matt, Matt Thomas, Maximillian Laumeister, mbrdev, Mellow, Michael Mairegger, Michael Richardson, Michał Perłakowski, mike z, Minhas Kamal, Mitch Talmadge, Mohammad Mirmostafa, Mr.Mindor, mshsayem, MuiBienCarlota, Nate Barbettini, Nicholas Sizer , nik, nollidge, Nuri Tasdemir, Oliver Mellet, Orlando William, Osama AbuSitta, Panda, Parth Patel, Patrick, Pavel Voronin, PSN, qJake, QoP, Racil Hilan, Radouane ROUFID, Rahul Nikate, Raidri, Rajeev, Rakitić, ravindra, rdans, Reeven, Richa Garg, Richard, Rion Williams, Rob, Robban, Robert Columbia, Ryan Hilbert, ryanyuyu, Sam, Sam Axe, Samuel, Sender, Shekhar, Shoe, Slayther, solidcell,

		Squidward, Squirrel, stackptr, stark, Stilgar, Sunny R Gupta, Suren Srapyan, Sworgkh, syb0rg, takrl, Tamir Vered, Theodoros Chatzigiannakis, Timothy Shields, Tom Droste, Travis J, Trent, Trikaldarshi, Troyen, Tushar patel, tzachs, Uri Agassi, Uriil, uTeisT, vcsjones, Ven, viggity, Vishal Madhvani, Vlad, Wai Ha Lee, Xiaoy312, Yury Kerbitskov, Zano, Ze Rubeus, Zimm1
21	C # 7.0 Caratteristiche	Adil Mammadov, afuna, Amitay Stern, Amr Badawy, Andreas Pähler , Andrew Diamond, Avi Turner, Benjamin Hodgson, Blorgbeard, bluray, Botond Balázs, Bovaz, Cerbrus, Clueless, Conrad.Dean, Dale Chen, David Pine, Degusto, Didgeridoo, Diligent Key Presser, ECC-Dan, Emre Bolat, fallaciousreasoning, ferday, Florian Greinacher, ganchito55, Ginkgo, H. Pauwelyn, Henrik H, Icy Defiance, Igor Ševo, iliketocode, Jatin Sanghvi, Jean-Bernard Pellerin, Jesse Williams, Jon Schoning, Kimmax, Kobi, Kris Vandermotten, Kritner, leppie, Llwyd, Maakep, maf-soft, Marc Gravell, MarcinJuraszek, Mariano Desanze, Matt Rowland, Matt Thomas, MemphiZ,

		mnoronha, MotKohn, Name, Nate Barbettini, Nico, Niek, nietras, NikolayKondratyev, Nuri Tasdemir, PashaPash, Pavel Mayorov, PeteGO, petrzjunior, Philippe, Pratik, Priyank Gadhiya, Pyritie, qJake, Raidri, Rakitić, RamenChef, Ray Vega, RBT, René Vogt, Rob, samuelesque , Squidward, Stavm, Stefano, Stefano d'Antonio, Stilgar, Tim Pohlmann, Uriil, user1304444, user2321864, user3185569, uTeisT, Uwe Keim, Vlad, Vlad, Wai Ha Lee, Wasabi Fan , WerWet, wezten, Wojciech Czerniak, Zze
22	C # Script	mehrandvd, Squidward, Stephen Leppik
23	caching	Aliaksei Futryn, th1rdey3
24	Classe e metodi parziali	Ben Jenkinson, Jonas S, Rahul Nikate, Stephen Leppik, Taras, The_Outsider
25	Classi statiche	MCronin, The_Outsider, Xiaoy312
26	CLSCompliantAttribute	mybirthname, Rob
27	Codice Contratti e asserzioni	Roy Dictus
28	Codice non sicuro in .NET	Andrew Piliser, cbale, codekaizen, Danny Varod, Isac, Jaroslav Kadlec, MSE, Nisarg Shah, Rahul Nikate, Stephen Leppik, Uwe Keim, ZenLulz

29	Come utilizzare C # Structs per creare un tipo Union (simile a C Unions)	DLeh, Milton Hernandez, Squidward, usr
30	Commenti e regioni	Bad, Botond Balázs, Jonathan Zúñiga, MrDKOz, Ranjit Singh, Squidward
31	Commenti sulla documentazione XML	Alexander Mandt, James , jHilscher, Jon Schneider, Nathan Tuggy, teo van kot, tsjnsn
32	Comprese le risorse di carattere	Bales, Facebamm
33	Contesto di sincronizzazione in attesa asincrona	codeape, Mark Shevchenko, RamenChef
34	Contratti di codice	MegaTron
35	Convenzioni di denominazione	Ben Aaronson, Callum Watkins, PMF, ZenLulz
36	Costrutti di flusso di dati di Task Parallel Library (TPL)	Droritos, Stephen Leppik
37	Costruttori e Finalizzatori	Adam Sills, Adi Lester, Adriano Repetti, Andrei Rînea, Andrew Diamond, Arjan Einbu, Avia, BackDoorNoBaby, BanksySan, Ben Fogel, Benjamin Hodgson, Benjol, Bogdan Gavril, Bovaz, Carlos Muñoz, Dan Hulme, Daryl, DLeh, Dmitry Bychenko, drusellers, Ehsan Sajjad, Fernando Matsumoto, guntbert, hatchet, Ian, Jeremy Kato, Jon Skeet, Julien Roncaglia, kamilk, Konamiman, ltiveron, Michael Richardson, Neel, Oly, Pavel Mayorov, Pavel Sapehin, Pavel Voronin, Peter

		Hommel, pinkfloydx33, Robert Columbia, RomCoo, Roy Dictus, Sam, Saravanan Sachi, Seph, Sklivvz, The_Cthulhu_Kid, Tim Medora, usr, Verena Haunschmid, void, Wouter, ZenLulz
38	Creazione del proprio MessageBox nell'applicazione Windows Form	Mansel Davies, Vaibhav_Welcomes_You
39	Creazione di un'applicazione console utilizzando un Editor di testo semplice e il compilatore C # (csc.exe)	delete me
40	Crittografia (System.Security.Cryptography)	glaubergft, MikeS159, Ogglas, Pete
41	Cronometri	Adam, demonplus, dotctor, Gavin Greenwalt , Jeppe Stig Nielsen, Sondre
42	Diagnostica	Jasmin Solanki, Luke Ryan, TylerH
43	Dichiarazioni condizionali	Alexander Mandt, Ameya Deshpande, EJoshuaS, H. Pauwelyn, Hayden, Kroltan, RamenChef, Sklivvz
44	Direttive preprocessore	Andrei, Gilad Naaman, Matas Vaitkevicius, qJake, RamenChef, theB, volvis
45	enum	Aaron Hudon, Abdul Rehman Sayed, Adrian Iftode, aholmes, alex, Blachshma, Chris Oldwood, Diligent Key Presser, dlatikay, Dmitry Bychenko, dove, Ghost4Man, H. Pauwelyn, ja72, Jon Schneider, Kit, konkked,

		Kyle Trauberman, Martin Zikmund, Matthew Whited, Maxime, mbrdev , Michael Mairegger, MuiBienCarlota, NikolayKondratyev, Osama AbuSitta, PSGuy , recursive, Richa Garg, Richard, Rob, sdgfsdh, Sergii Lischuk, Squirrel, Stefano d'Antonio, Tanner Swett, TarkaDaal , Theodoros Chatzigiannakis, vesi, Wasabi Fan, Yanai
46	Eredità	Almir Vuk, andre_ss6, Andrew Diamond, Barathon, Ben Aaronson , Ben Fogel, Benjol, David L, deloreyk, Ehsan Sajjad, harriyott, ja72, Jon Ericson, Karthik, Konamiman, MarcE, Matas Vaitkevicius, Pete Uh, Rion Williams, Robert Columbia, Steven , Suren Srapyan, VirusParadox, Yehuda Shapira
47	Esecuzione di richieste HTTP	Gordon Bell, Jon Schneider, Mark Shevchenko
48	Esempi Async / Waitit, Backgroundworker, Task e Thread	Dieter Meemken, Kyrylo M, nik, Pavel Mayorov, sebingel, Underscore, Xander Luciano, Yehor Hromadskyi
49	Espressioni Lambda	Andrei Rînea, Benjamin Hodgson, Benjol, David L, David Pine, Federico Allocati, Feelbad Soussi Wolfgun DZ, Fernando Matsumoto, H. Pauwelyn , haim770, Matas

		Vaitkevicius, Matt Sherman, Michael Mairegger, Michael Richardson, NotEnoughData, Oly, RubberDuck, S.L. Barth, Sunny R Gupta, Tagc, Thriggle
50	eventi	Aaron Hudon, Adi Lester , Benjol, CheGuevarasBeret, dcastro, matteeyah, meJustAndrew, mhoward, nik, niksofteng , NotEnoughData, OliPro007, paulius_l, PSGuy, Reza Aghaei, Roy Dictus, Squidward, Steven, vbnet3d
51	File e streaming I / O	BanksySan, Blachshma, dbmuller, DJCubed, Feelbad Soussi Wolfgun DZ, intox, Mikko Viitala, Sender, Squidward, Tolga Evcimen, Wasabi Fan
52	FileSystemWatcher	Sondre
53	Filtri di azione	Lokesh_Ram
54	Func delegati	Theodoros Chatzigiannakis, Valentin
55	Funzione con più valori di ritorno	Adam, Alexey Mitev, Durgpal Singh, Tolga Evcimen
56	Funzioni hash	Adi Lester, Callum Watkins, EvenPrime, ganchito55, Igor, jHilscher, RamenChef, ZenLulz
57	Garbage Collector in .Net	Andrei Rînea, da_sann, Eamon Charles, J3soon,

		Luke Ryan, Squidward, Suren Srapyan
58	Generare numeri casuali in C #	A. Can Aydemir, Adi Lester, Alexander Mandt , DLeh, J3soon, Rob
59	Generatore di query Lambda generico	4444, PedroSouki
60	Generazione del codice T4	lloyd, Pavel Mayorov
61	Generics	AGB, andre_ss6, Ben Aaronson, Benjamin Hodgson, Benjol, Bobson, Carsten, darth_phoenixx, dymanoid, Eamon Charles, Ehsan Sajjad, Gajendra, GregC, H. Pauwelyn, ja72, Jim, Kroltan, Matas Vaitkevicius, mehmetgil, meJustAndrew, Mord Zuber, Mujassir Nasir, Oly, Pavel Voronin, Richa Garg, Sam, Sebi, Sjoerd222888, Theodoros Chatzigiannakis, user3185569, VictorB, void, Wallace Zhang
62	Gestione di FormatException durante la conversione di stringhe in altri tipi	Rakitić, un-lucky
63	Gestore dell'autenticazione C #	Abbas Galiyakotwala
64	getto	Benjamin Hodgson, MSE , RamenChef, StriplingWarrior
65	guid	Bearington, Botond Balázs, elibyy, Jonas S, Osama AbuSitta, Sherantha, TarkaDaal, The_Outsider, Tim Ebenezer, void
66	I delegati	Aaron Hudon, Adam,

		Ben Aaronson, Benjamin Hodgson, Bradley Uffner, CalmBit, Cihan Yakar, CodeWarrior, EyasSH, Huseyin Durmus, Jasmin Solanki, Jeppe Stig Nielsen, Jon G, Jonas S, Matt, NikolayKondratyev, niksofteng, Rajput, Richa Garg, Sam Farajpour Ghamari, Shog9, Stu, Thulani Chivandikwa, trashr0x
67	ICloneable	ja72, Rob
68	IComparable	alex
69	Identità ASP.NET	HappyCoding, Skullomania
70	IEnumerable	4444, Avia, Benjamin Hodgson, Luke Ryan, Olivier De Meulder, The_Outsider
71	ILGenerator	Aleks Andreev, thehennyy
72	Immutabilità	Boggin, Jon Schneider, Oluwafemi, Tim Ebenezer
73	Implementazione del modello di design Flyweight	Jan Bońkowski
74	Implementazione del pattern di progettazione di Decorator	Jan Bońkowski
75	Implementazione di Singleton	Aaron Hudon, Adam, Adi Lester, Andrei Rînea, cbale, Disk Crasher, Ehsan Sajjad, Krzysztof Branicki, lothlarias, Mark Shevchenko, Pavel Mayorov, Sklivvz, snickro, Squidward, Squirrel, Stephen Leppik, Victor Tomaili, Xandrmoro

76	Importa contatti Google	4444, Supraj v
77	indicizzatore	A_Arnold, Ehsan Sajjad, jHilscher
78	Iniezione di dipendenza	Buh Buh, iaminvinicble, Kyle Trauberman, Wiktor Dębski
79	Inizializzatori di oggetti	Andrei, Kroltan, LeopardSkinPillBoxHat, Marco, Nick DeVore, Stephen Leppik
80	Inizializzatori di raccolta	Aphelion, ASh, Bart Jolling, Chronocide, CodeCaster, CyberFox, DLeh, Jacob Linney, Jeromy Irvine, Jonas S, Matas Vaitkevicius, Rob, robert demartino, rudygt, Squidward, Tamir Vered, TarkaDaal, Thulani Chivandikwa, WMios
81	Inizializzazione delle proprietà	Blorgbeard, hatchet, jaycer, Michael Sorens, Parth Patel, Stephen Leppik
82	interfacce	Avia, Botond Balázs, CyberFox, harriyott, hellyale, Jeremy Kato, MarcE, MSE, PMF, Preston, Sigh, Sometowngeek, Stagg, Steven, user2441511
83	Interfaccia IDisposable	Aaron Hudon, Adam, BatteryBackupUnit, binki, Bogdan Gavril, Bryan Crosby, ChrisWue, Dmitry Bychenko, Ehsan Sajjad, H. Pauwelyn, Jarrod Dixon, Josh Peterson, Matas Vaitkevicius, Maxime, Nicholas Sizer,

		OliPro007, Pavel Mayorov, pinkfloydx33, pyrocumulus, RamenChef, Rob, Thennarasan, Will Ray
84	Interfaccia INotifyPropertyChanged	mbrdev, Stephen Leppik, Vlad
85	Interfaccia IQueryable	lucavgobbi, Michiel van Oosterhout, RamenChef, Rob
86	interoperabilità	Balen Danny, Benjamin Hodgson, Bovaz, Craig Brett, Dean Van Greunen, Gajendra, Jan Bońkowski, Kimmax, Marc Wittmann, Martin, Pavel Durov, René Vogt, RomCoo, Squidward
87	Interpolazione a stringa	Arjan Einbu, ATechieThought, avs099 , bluray, Brendan L, Dave Zych, DLeh, Ehsan Sajjad, fabriciorissetto, Guilherme de Jesus Santos, H. Pauwelyn, Jon Skeet, Nate Barbettini, RamenChef, Rion Williams, Squidward, Stephen Leppik, Tushar patel, Wasabi Fan
88	iteratori	Botond Balázs, Lijo, Nate Barbettini, Tagc
89	La gestione delle eccezioni	0x49D1, Abdul Rehman Sayed, Adam Lear, Adil Mammadov, Andrew Diamond, Aseem Gautam, Athafoud, Botond Balázs, Collin Stevens, Danny Chen, Dmitry Bychenko, dove, Eldar Dordzhiev,

		fabriciorissetto, faso, flq, George Duckett, Gilad Naaman, Gudradain, Jack, James Hughes, Jamie Rees, John Meyer , Jonesopolis, MadddinTribleD, Marimba, Matas Vaitkevicius, Matt, matteeyah, Mendhak, Michael Bisbjerg, Nate Barbettini, Nathaniel Ford, nik0lias, niksofteng , Oly, Pavel Pája Halbich , Pavel Voronin, PMF, Racil Hilan, raidensan, Rasa , Robert Columbia, RomCoo, Sam Hanley, Scott Koland, Squidward , Steve Dunn, Thulani Chivandikwa, vesi
90	Lambda Expressions	H. Pauwelyn, Oly
91	Le tuple	Bovaz, Chawin, EFrank, H. Pauwelyn, Mark Benovsky, Muhammad Albarmawi, Nathan Tuggy, Nikita, Nuri Tasdemir, petrzjunior, PMF, RaYell, slawekwin, Squidward, tire0011
92	Leggi e capisci Stacktraces	S.L. Barth
93	letterali	jaycer, NotEnoughData, Racil Hilan
94	Lettura e scrittura di file .zip	4444, DLeh, Naveen Gogineni, Nisarg Shah
95	LINQ in XML	Denis Elkhov, Stephen Leppik, Uali
96	LINQ parallelo (PLINQ)	Adi Lester
97	Linq to Objects	brijber, Christian Gollhardt, FortyTwo, Kevin Green, Raphael

		Pantaleão, Simon Halsey, Tanveer Badar
98	Lock Statement	Aaron Hudon, Alexey Groshev, Andrei Rînea, Benjamin Hodgson, Botond Balázs, Christopher Currens, Cihan Yakar, David Ben Knoble, Denis Elkhov, Diligent Key Presser, George Duckett, George Polevoy, Jargon, Jasmin Solanki, Jivan, Mark Shevchenko, Matas Vaitkevicius, Mikko Viitala, Nuri Tasdemir, Oluwafemi, Pavel Mayorov, Richard, Rob, Scott Hannen, Squidward, Vahid Farahmandian
99	looping	Alisson, Andrei Rînea, B Hawkins, Benjamin Hodgson, Botond Balázs , connor, Dialecticus, DJCubed, Freelex, Jon Schneider, Oluwafemi, Racil Hilan, Squidward, Testing123, Tolga Evcimen
100	Manipolazione delle stringhe	Blachshma, Jon Schneider, sferencik, The_Outsider
101	metodi	Botz3000, F_V, fubo, H. Pauwelyn, Icy Defiance, Jasmin Solanki, Jeremy Kato, Jon Schneider, ken2k, Marco, meJustAndrew, MSL, S.Dav, Sjoerd222888, TarkaDaal, un-lucky
102	Metodi DateTime	AbdulRahman Ansari, C4u, Christian Gollhardt,

		Felipe Oriani, Guilherme de Jesus Santos, James Hughes, Matas Vaitkevicius, midnightsyntax, Mostafiz , Oluwafemi, Pavel Yermalovich, Sondre, theinarasu, Thulani Chivandikwa
103	Metodi di estensione	Aaron Hudon, AbdulRahman Ansari, Adi Lester, Adil Mammadov, AGB, AldoRomo88, anaximander, Aphelion, Ashwin Ramaswami, ATechieThought, Ben Aaronson, Benjol, binki, Bjørn-Roger Kringsjå, Blachshma, Blorgbeard, Brett Veenstra, brijber, Callum Watkins, Chad McGrath, Charlie H, Chris Akridge, Chronocide, CorrectorBot, cubrr, Dan-Cook, Daniel Stradowski, David G., David Pine, Deepak gupta, diiN__________, DLeh, Dmitry Bychenko, DoNot, DWright, Đan, Ehsan Sajjad, ekolis, el2iot2, Elton, enrico.bacis, Erik Schierboom, ethorn10, extremeboredom, Ezra, fahadash, Federico Allocati, Fernando Matsumoto, FrankerZ, gdziadkiewicz, Gilad Naaman, GregC, Gudradain, H. Pauwelyn, HimBromBeere, Hsu Wei Cheng, Icy Defiance, Jamie Rees, Jeppe Stig

		Nielsen, John Peters, John Slegers, Jon Erickson, Jonas S, Jonesopolis, Kev, Kevin Avignon, Kevin DiTraglia , Kobi, Konamiman, krillgar, Kurtis Beavers, Kyle Trauberman, Lafexlos, LMK, lothlarias, Lukáš Lánský, Magisch, Marc, MarcE, Marek Musielak, Martin Zikmund, Matas Vaitkevicius, Matt, Matt Dillard, Maximilian Ast, mbrdev, MDTech.us_MAN, meJustAndrew, Michael Benford, Michael Freidgeim, Michael Richardson, Michał Perłakowski, Nate Barbettini, Nick Larsen, Nico, Nisarg Shah, Nuri Tasdemir, Parth Patel, pinkfloydx33, PMF, Prashanth Benny, QoP, Raidri, Reddy, Reeven, Ricardo Amores, Richard , Rion Williams, Rob, Robert Columbia, Ryan Hilbert, ryanyuyu, S. Tarık Çetin, Sam Axe, Shoe , Sibeesh Venu, solidcell , Sondre, Squidward, Steven, styfle, SysVoid, Tanner Swett, Timothy Rascher, TKharaishvili, T-moty, Tobbe, Tushar patel, unarist, user3185569, user40521 , Ven, Victor Tomaili, viggity
104	Microsoft.Exchange.WebServices	Bassie
105	Modelli di design creativo	DWright, Jan Bońkowski,

		Mark Shevchenko, Parth Patel, PedroSouki, Pierre Theate, Sondre, Tushar patel
106	Modelli di progettazione strutturale	Timon Post
107	Modificatori di accesso	Botond Balázs, H. Pauwelyn, hatcyl, John, Justin Rohr, Kobi, Robert Woods, Thaoden, ZenLulz
108	Networking	Adi Lester, Nicholaus Lawson, Salih Karagoz, shawty, Squirrel, Xander Luciano
109	nome dell'operatore	Chad, Danny Chen, heltonbiker, Kane, MotKohn, Philip C, pinkfloydx33, Racil Hilan, Rob, Robert Columbia, Sender, Sondre, Stephen Leppik, Wasabi Fan
110	Nullable types	Benjamin Hodgson, Braydie, DmitryG, Gordon Bell, Jasmin Solanki, Jon Schneider, Konstantin Vdovkin, Maximilian Ast, Mikko Viitala, Nicholas Sizer, Patrick Hofman, Pavel Mayorov, pinkfloydx33, Vitaliy Fedorchenko
111	NullReferenceException	4444, Agramer, Ashutosh, krimog, Kyle Trauberman, Mathias Müller, Philip C, RamenChef, S.L. Barth, Shelby115, Squidward, vicky, Zikato
112	O (n) Algoritmo per la rotazione circolare di un array	AFT
113	ObservableCollection	demonplus, GeralexGR,

		Jonathan Anctil, MuiBienCarlota
114	Operatore di uguaglianza	Vadim Martynov
115	Operatore Null Coalescing	aashishkoirala, Ankit Rana, Aristos, Bradley Uffner, David Arno, David G., David Pine, demonplus, Denis Elkhov, Diligent Key Presser, Eamon Charles, Ehsan Sajjad, eouw0o83hf, Fernando Matsumoto, H. Pauwelyn , Jodrell, Jon Schneider, Jonesopolis, Martin Zikmund, Mike C, Nate Barbettini, Nic Foster, petelids, Prateek, Rahul Nikate, Rion Williams, Rob, smead, tonirush, Wasabi Fan, Will Ray
116	operatori	Adam Houldsworth, Adi Lester, Adil Mammadov, Akshay Anand, Alan McBee, Avi Turner, Ben Fogel, Blorgbeard, Blubberguy22, Chris Jester-Young, David Basarab, DLeh, Dmitry Bychenko, dotctor, Ehsan Sajjad, fabriciorissetto, Fernando Matsumoto, H. Pauwelyn, Henrik H, Jake Farley, Jasmin Solanki, Jephron, Jeppe Stig Nielsen, Jesse Williams, Joe, JohnLBevan, Jon Schneider, Jonas S, Kevin Montrose, Kimmax , lokusking, Matas Vaitkevicius, meJustAndrew, Mikko Viitala, mmushtaq,

		Mohamed Belal, Nate Barbettini, Nico, Oly, pascalhein, Pavel Voronin, petelids, Philip C, Racil Hilan, RhysO, Robert Columbia, Rodolfo Fadino Junior, Sachin Joseph, Sam, slawekwin, slinzerthegod , Squidward, Testing123, TyCobb, Wasabi Fan, Xiaoy312, Zaheer Ul Hassan
117	Operatori non condizionali	Alpha, dazerdude, DLeh, Draken, George Duckett, Jon Schneider, Kobi, Max, Nathan, Nicholas Sizer, Rob, Stephen Leppik, tehDorf, Timothy Shields, topolm, Wasabi Fan
118	Operazioni stringhe comuni	Austin T French, Blachshma, bluish, CharithJ, Chief Wiggum, cyberj0g, Daryl, deloreyk , jaycer, Jaydip Jadhav, Jon G, Jon Schneider, juergen d, Konamiman, Maniero, Paul Weiland, Racil Hilan, RoelF, Stefan Steiger, Steven, The_Outsider, tiedied61, un-lucky, WizardOfMenlo
119	Parola chiave rendimento	Aaron Hudon, Andrew Diamond, Ben Aaronson , ChrisPatrick, Damon Smithies, David G., David Pine, Dmitry Bychenko, dotctor, Ehsan Sajjad, erfanrazi, Gajendra, George Duckett, H. Pauwelyn, HimBromBeere, Jeremy Kato, João Lourenço, Joe Amenta, Julien

		Roncaglia, just.ru, Karthik AMR, Mark Shevchenko, Michael Richardson, MuiBienCarlota, Myster, Nate Barbettini, Noctis, Nuri Tasdemir, Olivier De Meulder, OP313, ravindra, Ricardo Amores, Rion Williams, rocky, Sompom, Tot Zam, un-lucky, Vlad, void, Wasabi Fan, Xiaoy312, ZenLulz
120	parole	4444, A_Arnold, Aaron Hudon, Ade Stringer, Adi Lester, Aditya Korti, Adriano Repetti, AJ., Akshay Anand, Alex Filatov, Alexander Pacha , Amir Pourmand, Andrei Rînea, Andrew Diamond, Angela, Anna, Avia, Bart , Ben, Ben Fogel, Benjamin Hodgson, Bjørn-Roger Kringsjå, Botz3000, Brandon, brijber, BrunoLM, BunkerMentality, BurnsBA, bwegs, Callum Watkins, Chris, Chris Akridge, Chris H., Chris Skardon, ChrisPatrick, Chuu, Cihan Yakar, cl3m , Craig Brett, Daniel, Daniel J.G., Danny Chen , Darren Davies, Daryl, dasblinkenlight, David, David G., David L, David Pine, DAXaholic, deadManN, DeanoMachino, digitlworld, Dmitry Bychenko, dotctor, DPenner1, Drew Kennedy, DrewJordan,

Ehsan Sajjad, EJoshuaS , Elad Lachmi, Eric Lippert, EvenPrime, F_V, Felix, fernacolo, Fernando Matsumoto, forsvarir, Francis Lord, Gavin Greenwalt, gdoron , George Duckett, Gilad Naaman, goric, greatwolf , H. Pauwelyn, Happypig375, Icemanind , Jack, Jacob Linney, Jake, James Hughes, Jcoffman, Jeppe Stig Nielsen, jHilscher, João Lourenço, John Slegers, JohnD, Jon Schneider, Jon Skeet, JoshuaBehrens, Kilazur, Kimmax, Kirk Woll, Kit, Kjartan, kjhf, Konamiman , Kyle Trauberman, kyurthich, levininja, lokusking, Mafii, Mamta D, Mango Wong, MarcE, MarcinJuraszek, Marco Scabbiolo, Martin, Martin Klinke, Martin Zikmund, Matas Vaitkevicius, Mateen Ulhaq, Matěj Pokorný, Mat's Mug, Matthew Whited, Max, Maximilian Ast, Medeni Baykal, Michael Mairegger, Michael Richardson, Michel Keijzers, Mihail Shishkov , mike z, Mr.Mindor, Myster, Nicholas Sizer, Nicholaus Lawson, Nick Cox, Nico, nik, niksofteng, NotEnoughData, numaroth, Nuri Tasdemir , pascalhein, Pavel Mayorov, Pavel Pája Halbich, Pavel

		Yermalovich, Pavieł Kraskoŭski, Paweł Mach, petelids, Peter Gordon, Peter L., PMF, Rakitić, RamenChef, ranieuwe, Razan, RBT, Renan Gemignani, Ringil, Rion Williams, Rob, Robert Columbia, ro binmckenzie, RobSiklos, Romain Vincent, RomCoo, ryanyuyu, Sain Pradeep, Sam, Sándor Mátyás Márton, Sanjay Radadiya, Scott, sebingel, Skipper, Sobieck, sohnryang, somebody, Sondre, Squidward, Stephen Leppik, Sujay Sarma, Suyash Kumar Singh, svick, TarkaDaal, th1rdey3, Thaoden, Theodoros Chatzigiannakis, Thorsten Dittmar, Tim Ebenezer, titol, tonirush, topolm, Tot Zam, user3185569, Valentin, vcsjones, void, Wasabi Fan, Wavum, Woodchipper, Xandrmoro, Zaheer Ul Hassan, Zalomon, Zohar Peled
121	Per iniziare: Json con C #	Neo Vijay, Rob, VitorCioletti
122	Polimorfismo	Ade Stringer, ganchito55 , H. Pauwelyn, Karthik, Maximilian Ast, void
123	Presa asincrona	Timon Post
124	Programmazione funzionale	Andrei Epure, Boggin, Botond Balázs, richard

125	Programmazione orientata agli oggetti in C #	Yashar Aliabasi
126	Proprietà	Botond Balázs, Callum Watkins, Jeremy Kato, John, JohnLBevan, niksofteng, Stephen Leppik, Zohar Peled
127	puntatori	Jeppe Stig Nielsen, Theodoros Chatzigiannakis
128	Puntatori e codice non sicuro	Aaron Hudon, Botond Balázs, undefined
129	Query LINQ	Adam Clifford, Ade Stringer, Adi Lester, Adil Mammadov, Akshay Anand, Aleksey L., Alexey Koptyaev, AMW, anaximander, Andrew Piliser, Ankit Vijay, Aphelion, bbonch, Benjamin Hodgson, bmadtiger, BOBS, BrunoLM, Budi, bumbeishvili, callisto, cbale, Chad McGrath, Chris, Chris H., coyote, Daniel Argüelles, Daniel Corzo, darcyq, David, David G., David Pine, DavidG, die maus, Diligent Key Presser, Dmitry Bychenko, Dmitry Egorov, dotctor, Ehsan Sajjad, Erick, Erik Schierboom, EvenPrime, fabriciorissetto, faso, Finickyflame, Florin M, forsvarir, fubo, gbellmann, Gene, Gert Arnold, Gilad Green, H. Pauwelyn, Hari Prasad, hellyale, HimBromBeere, hWright, iliketocode, Ioannis Karadimas, Jagadisha B S, James

		Ellis-Jones, jao, jiaweizhang, Jodrell, Jon Bates, Jon G, Jon Schneider, Jonas S, karaken12, KevinM, Koopakiller, leppie, LINQ , Lohitha Palagiri, ltiveron, Mafii, Martin Zikmund, Matas Vaitkevicius, Mateen Ulhaq, Matt, Maxime, mburleigh, Meloviz, Mikko Viitala, Mohammad Dehghan, mok, Nate Barbettini, Neel, Neha Jain, Néstor Sánchez A., Nico, Noctis , Pavel Mayorov, Pavel Yermalovich, Paweł Hemperek, Pedro, Phuc Nguyen, pinkfloydx33, przno, qJake, Racil Hilan , rdans, Rémi, Rion Williams, rjdevereux, RobPethi, Ryan Abbott, S. Rangeley, S.Akbari, S.L. Barth, Salvador Rubio Martinez, Sanjay Radadiya, Satish Yadav, sebingel, Sergio Domínguez, SilentCoder, Sivanantham Padikkasu, slawekwin, Sondre, Squidward, Stephen Leppik, Steve Trout, Tamir Vered, techspider, teo van kot, th1rdey3, Theodoros Chatzigiannakis, Tim Iles , Tim S. Van Haren, Tobbe, Tom, Travis J, tungns304, Tushar patel, user1304444, user3185569, Valentin, varocarbas, VictorB, Vitaliy Fedorchenko, vivek nuna, void, wali,

		wertzui, WMios, Xiaoy312, Yaakov Ellis, Zev Spitz
130	Reactive Extensions (Rx)	stefankmitph
131	Regex Parsing	C4u
132	Rendere sicuro un thread variabile	Wyck
133	ricorsione	Alexey Groshev, Botond Balázs, connor, ephtee, Florian Koch, Kroltan, Michael Brandon Morris, Mulder, Pan, qJake, Robert Columbia, Roy Dictus, SlaterCodes, Yves Schelpe
134	Riflessione	Alexander Mandt, Aman Sharma, artemisart, Aseem Gautam, Axarydax, Benjamin Hodgson, Botond Balázs, Carson McManus, Cigano Morrison Mendez, Cihan Yakar, da_sann, DVJex, Ehsan Sajjad, H. Pauwelyn, Haim Bendanan, HimBromBeere, James Ellis-Jones, James Hughes, Jamie Rees, Jan Peldřimovský, Johny Skovdal, JSF, Kobi, Konamiman, Kristijan, Lovy, Matas Vaitkevicius, Mourndark, Nuri Tasdemir, pinkfloydx33, Rekshino, René Vogt, Sachin Chavan, Shuffler, Sjoerd222888, Sklivvz, Tamir Vered, Thriggle, Travis J, uygar.raf, Vadim Ovchinnikov, wablab, Wai Ha Lee
135	Risoluzione di sovraccarico	Dunno, Petr Hudeček,

		Stephen Leppik, TorbenJ
136	Runtime Compile	Artificial Stupidity, Stephen Leppik, Tommy
137	ruscello	Danny Bogers, jlawcordova, Jon Schneider, Nuri Tasdemir, Pushpendra
138	Selezionato e deselezionato	Botond Balázs, Rahul Nikate, Sam Johnson, ZenLulz
139	Sequenze di escape delle stringhe	Benjol, Botond Balázs, cubrr, Ed Gibbs, Jeppe Stig Nielsen, LegionMammal978, Michael Richardson, Peter Gordon, Petr Hudeček, Squidward, tonirush
140	Serializzazione binaria	David, Maxim, RamenChef, Stephen Leppik
141	straripamento	Akshay Anand, Nuri Tasdemir, tonirush
142	String Concatenate	Abdul Rehman Sayed, Callum Watkins, ChaoticTwist, Doruk, Dweeberly, Jon Schneider, Oluwafemi, Rob, RubberDuck, Testing123, The_Outsider
143	String.Format	Aaron Hudon, Akshay Anand, Alexander Mandt, Andrius, Aseem Gautam, Benjol, BrunoLM, Dmitry Egorov, Don Vince, Dweeberly, ebattulga, ejhn5, gdoron, H. Pauwelyn, Hossein Narimani Rad, Jasmin

		Solanki, Marek Musielak, Mark Shevchenko, Matas Vaitkevicius, Mendhak, MGB, nikchi, Philip C, Rahul Nikate, Raidri, RamenChef, Richard, Richard, Rion Williams, ryanyuyu, teo van kot, Vincent, void, Wyck
144	StringBuilder	ATechieThought, brijber, Jeremy Kato, Jon Schneider, Robert Columbia, The_Outsider
145	Stringhe Verbatim	Alan McBee, Amitay Stern, Andrew Diamond, Aphelion, Arjan Einbu, avb, Bryan Crosby, Charlie H, David G., devuxer, DLeh, Ehsan Sajjad, Freelex, goric, Jared Hooper, Jeremy Kato, Jonas S, Kevin Montrose, Kilazur, Mateen Ulhaq, Ricardo Amores, Rion Williams, Sam Johnson, Sophie Jackson-Lee, Squirrel, th1rdey3
146	Structs	abto, Alexey Groshev, Benjamin Hodgson, Botz3000, David, Elad Lachmi, ganchito55, Jon Schneider, NikolayKondratyev
147	System.DirectoryServices.Protocols.LdapConnection	Andrew Stollak
148	System.Management.Automation	Mikko Viitala
149	Task Libreria parallela	Benjamin Hodgson, Brandon, Collin Stevens, i3arnon, Mokhtar Ashour, Murtuza Vohra

150	threading	Aaron Hudon, Alexander Petrov, Austin T French, captainjamie, Eldar Dordzhiev, H. Pauwelyn, ionmike, Jacob Linney, JohnLBevan, leondepdelaw, Mamta D, Matthijs Wessels, Mellow , RamenChef, Zoba
151	Timer	Adam, Akshay Anand, Benjamin Kozuch, ephtee, RamenChef, Thennarasan
152	Tipi anonimi	Fernando Matsumoto, goric, Stephen Leppik
153	Tipi incorporati	Alexander Mandt, David, F_V, Haseeb Asif, matteeyah, Patrick Hofman, Wai Ha Lee
154	Tipo di conversione	Community, connor, Ehsan Sajjad, Lijo
155	Tipo di valore vs Tipo di riferimento	Abdul Rehman Sayed, Adam, Amir Pourmand, Blubberguy22, Chronocide, Craig Brett, docesam, GWigWam, matiaslauriti, meJustAndrew, Michael Mairegger, Michele Ceo, Moe Farag, Nate Barbettini, RamenChef, Rob, scher, Snympi, Tagc, Theodoros Chatzigiannakis
156	Tipo dinamico	Daryl, David, H. Pauwelyn, Kilazur, Mark Shevchenko, Nate Barbettini, Rob
157	Uguale e GetHashCode	Alexey, BanksySan, hatcyl, ja72, Jeppe Stig Nielsen, meJustAndrew,

		Rob, scher, Timitry, viggity
158	Una panoramica delle collezioni c #	Aaron Hudon, Andrew Diamond, Denuath, Jeremy Kato, Jon Schneider, Jorge, Juha Palomäki, Leon Husmann, Michael Mairegger, Michael Richardson, Nikita, rene, Rob, Sebi, TarkaDaal, wertzui, Will Ray
159	Uso della direttiva	Fernando Matsumoto, Jesse Williams, JohnLBevan, Kit, Michael Freidgeim, Nuri Tasdemir, RamenChef, Tot Zam
160	Utilizzando json.net	Aleks Andreev, Snipzwolf
161	Utilizzando la dichiarazione	Adam Houldsworth, Ahmar, Akshay Anand, Alex Wiese, andre_ss6, Aphelion, Benjol, Boris Callens, Bradley Grainger, Bradley Uffner, bubbleking, Chris Marisic , ChrisWue, Cristian T, cubrr, Dan Ling, Danny Chen, dav_i, David Stockinger, dazerdude, Denis Elkhov, Dmitry Bychenko, Erik Schierboom, Florian Greinacher, gdoron, H. Pauwelyn, Herbstein, Jon Schneider, Jon Skeet, Jonesopolis, JT., Ken Keenan, Kev, Kobi, Kyle Trauberman, Lasse Vågsæther Karlsen, LegionMammal978, Lorentz Vedeler, Martin, Martin Zikmund, Maxime

		, Nuri Tasdemir, Peter K, Philip C, pid, René Vogt, Rion Williams, Ryan Abbott, Scott Koland, Sean, Sparrow, styfle, Sunny R Gupta, Sworgkh, Thaoden, The_Cthulhu_Kid, Tom Droste, Tot Zam, Zaheer Ul Hassan
162	Utilizzo di SQLite in C #	Carmine, NikolayKondratyev, th1rdey3, Tim Yusupov
163	Windows Communication Foundation	NtFreX
164	XDocument e lo spazio dei nomi System.Xml.Linq	Crowcoder, Jon Schneider
165	XmlDocument e lo spazio dei nomi System.Xml	Alexander Petrov, Rokey Ge, Rubens Farias, Timon Post, Willy David Jr