EBook Gratuito

APPRENDIMENTO .NET Framework

Free unaffiliated eBook created from
Stack Overflow contributors.

#.net

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: -net-framework

It is an unofficial and free .NET Framework ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official .NET Framework.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con .NET Framework

Osservazioni

.NET Framework è un set di librerie e un runtime, originariamente progettato da Microsoft. Tutti i programmi .NET vengono compilati in un bytecode denominato MSIL (Microsoft Intermediate Language). MSIL viene eseguito dal Common Language Runtime (CLR).

Di seguito è possibile trovare diversi esempi di "Hello World" in varie lingue che supportano .NET Framework. "Hello World" è un programma che visualizza "Hello World" sul dispositivo di visualizzazione. È usato per illustrare la sintassi di base per la costruzione di un programma di lavoro. Può anche essere utilizzato come test di integrità per garantire che il compilatore, l'ambiente di sviluppo e l'ambiente di runtime di una lingua funzionino correttamente.

Elenco delle lingue supportate da .NET

Versioni

.NETTO

Versione	Data di rilascio
1.0	2002/02/13
1.1	2003/04/24
2.0	2005-11-07
3.0	2006-11-06
3.5	2007-11-19
3.5 SP1	2008-08-11
4.0	2010-04-12
4.5	2012-08-15
4.5.1	2013/10/17
4.5.2	2014/05/05
4.6	2015/07/20
4.6.1	2015/11/17
4.6.2	2016/08/02

Versione	Data di rilascio
4.7	2017/04/05

Quadro compatto

Versione	Data di rilascio
1.0	2000-01-01
2.0	2005-10-01
3.5	2007-11-19
3.7	2009-01-01
3.9	2013/06/01

Micro Framework

Versione	Data di rilascio
4.2	2011-10-04
4.3	2012/12/04
4.4	2015/10/20

Examples

Ciao mondo in C

```
using System;

class Program
{
    // The Main() function is the first function to be executed in a program
    static void Main()
    {
        // Write the string "Hello World to the standard out
        Console.WriteLine("Hello World");
    }
}
```

`Console.WriteLine` ha diversi overload. In questo caso, la stringa "Hello World" è il parametro e produrrà "Hello World" sullo stream standard durante l'esecuzione. Altri sovraccarichi possono chiamare `.ToString` dell'argomento prima di scrivere nello stream. Vedere la documentazione di

.NET Framework per ulteriori informazioni.

Demo live in azione su .NET Fiddle

Introduzione a C #

Ciao mondo in Visual Basic .NET

```
Imports System

Module Program
    Public Sub Main()
        Console.WriteLine("Hello World")
    End Sub
End Module
```

Demo live in azione su .NET Fiddle

Introduzione a Visual Basic .NET

Ciao mondo in F

```
open System

[<EntryPoint>]
let main argv =
    printfn "Hello World"
    0
```

Demo live in azione su .NET Fiddle

Introduzione a F #

Ciao mondo in C ++ / CLI

```
using namespace System;

int main(array<String^>^ args)
{
    Console::WriteLine("Hello World");
}
```

Hello World in PowerShell

```
Write-Host "Hello World"
```

Introduzione a PowerShell

Ciao mondo a Nemerle

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

Ciao mondo in Oxygene

```
namespace HelloWorld;

interface

type
  App = class
  public
    class method Main(args: array of String);
  end;

implementation

class method App.Main(args: array of String);
begin
  Console.WriteLine('Hello World');
end;

end.
```

Ciao mondo in Boo

```
print "Hello World"
```

Hello World in Python (IronPython)

```
print "Hello World"
```

```
import clr
from System import Console
Console.WriteLine("Hello World")
```

Ciao mondo in IL

```
.class public auto ansi beforefieldinit Program
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Main() cil managed
  {
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello World"
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  }

  .method public hidebysig specialname rtspecialname
         instance void  .ctor() cil managed
  {
```

```
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  }

}
```

Leggi Iniziare con .NET Framework online: https://riptutorial.com/it/dot-net/topic/14/iniziare-con--net-framework

Capitolo 2: .NET Core

introduzione

.NET Core è una piattaforma di sviluppo generica gestita da Microsoft e dalla comunità .NET su GitHub. È multipiattaforma, supporta Windows, macOS e Linux e può essere utilizzata in scenari dispositivo, cloud e embedded / IoT.

Quando pensi a .NET Core, tieni presente quanto segue (implementazione flessibile, strumenti multipiattaforma, da riga di comando, open source).

Un'altra cosa fantastica è che anche se è open source, Microsoft lo sta attivamente supportando.

Osservazioni

Di per sé, .NET Core include un singolo modello di applicazione - app per console - utile per strumenti, servizi locali e giochi basati su testo. Ulteriori modelli applicativi sono stati creati su .NET Core per estenderne le funzionalità, come ad esempio:

- ASP.NET Core
- Windows 10 Universal Windows Platform (UWP)
- Xamarin.Forms

Inoltre, .NET Core implementa la libreria standard .NET e pertanto supporta le librerie standard .NET.

.NET Standard Library è una specifica API che descrive il set coerente di API .NET che gli sviluppatori possono aspettarsi in ogni implementazione .NET. Le implementazioni .NET devono implementare questa specifica per poter essere considerate compatibili con la libreria standard .NET e per supportare librerie destinate alla libreria standard .NET.

Examples

App Console di base

```
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("\nWhat is your name? ");
        var name = Console.ReadLine();
        var date = DateTime.Now;
        Console.WriteLine("\nHello, {0}, on {1:d} at {1:t}", name, date);
        Console.Write("\nPress any key to exit...");
        Console.ReadKey(true);
    }
}
```

Leggi .NET Core online: https://riptutorial.com/it/dot-net/topic/9059/-net-core

Capitolo 3: ADO.NET

introduzione

ADO (ActiveX Data Objects) .Net è uno strumento fornito da Microsoft che fornisce l'accesso a origini dati quali SQL Server, Oracle e XML attraverso i suoi componenti. Le applicazioni front-end .Net possono recuperare, creare e manipolare i dati, una volta connessi a un'origine dati tramite ADO.Net con i privilegi appropriati.

ADO.Net fornisce un'architettura senza connessione. È un approccio sicuro per interagire con un database, poiché la connessione non deve essere mantenuta durante l'intera sessione.

Osservazioni

Una nota sulla parametrizzazione di SQL con `Parameters.AddWithValue` : `AddWithValue` non è mai un buon punto di partenza. Questo metodo si basa sulla deduzione del tipo di dati da ciò che viene passato. Con questo, si potrebbe finire in una situazione in cui la conversione impedisce alla query di utilizzare un indice . Si noti che alcuni tipi di dati di SQL Server, come `char` / `varchar` (senza precedente "n") o `date` , non hanno un tipo di dati .NET corrispondente. In questi casi, è necessario utilizzare `Add` con il tipo di dati corretto .

Examples

Esecuzione di istruzioni SQL come comando

```
// Uses Windows authentication. Replace the Trusted_Connection parameter with
// User Id=...;Password=...; to use SQL Server authentication instead. You may
// want to find the appropriate connection string for your server.
string connectionString =
@"Server=myServer\myInstance;Database=myDataBase;Trusted_Connection=True;"

string sql = "INSERT INTO myTable (myDateTimeField, myIntField) " +
    "VALUES (@someDateTime, @someInt);";

// Most ADO.NET objects are disposable and, thus, require the using keyword.
using (var connection = new SqlConnection(connectionString))
using (var command = new SqlCommand(sql, connection))
{
    // Use parameters instead of string concatenation to add user-supplied
    // values to avoid SQL injection and formatting issues. Explicitly supply datatype.

    // System.Data.SqlDbType is an enumeration. See Note1
    command.Parameters.Add("@someDateTime", SqlDbType.DateTime).Value = myDateTimeVariable;
    command.Parameters.Add("@someInt", SqlDbType.Int).Value = myInt32Variable;

    // Execute the SQL statement. Use ExecuteScalar and ExecuteReader instead
    // for query that return results (or see the more specific examples, once
    // those have been added).

    connection.Open();
```

```
    command.ExecuteNonQuery();
}
```

Nota 1: vedere Enumerazione SqlDbType per la variazione specifica di MSFT SQL Server.

Nota 2: Si prega di consultare Enumerazione MySqlDbType per la variazione specifica di MySQL.

Best practice: esecuzione di istruzioni SQL

```
public void SaveNewEmployee(Employee newEmployee)
{
    // best practice - wrap all database connections in a using block so they are always
closed & disposed even in the event of an Exception
    // best practice - retrieve the connection string by name from the app.config or
web.config (depending on the application type) (note, this requires an assembly reference to
System.configuration)
    using(SqlConnection con = new
SqlConnection(System.Configuration.ConfigurationManager.ConnectionStrings["MyConnectionName"].Connectio

    {
        // best practice - use column names in your INSERT statement so you are not dependent
on the sql schema column order
        // best practice - always use parameters to avoid sql injection attacks and errors if
malformed text is used like including a single quote which is the sql equivalent of escaping
or starting a string (varchar/nvarchar)
        // best practice - give your parameters meaningful names just like you do variables in
your code
        using(SqlCommand sc = new SqlCommand("INSERT INTO employee (FirstName, LastName,
DateOfBirth /*etc*/) VALUES (@firstName, @lastName, @dateOfBirth /*etc*/)", con))
        {
            // best practice - always specify the database data type of the column you are
using
            // best practice - check for valid values in your code and/or use a database
constraint, if inserting NULL then use System.DbNull.Value
            sc.Parameters.Add(new SqlParameter("@firstName", SqlDbType.VarChar, 200){Value =
newEmployee.FirstName ?? (object) System.DBNull.Value});
            sc.Parameters.Add(new SqlParameter("@lastName", SqlDbType.VarChar, 200){Value =
newEmployee.LastName ?? (object) System.DBNull.Value});

            // best practice - always use the correct types when specifying your parameters,
Value is assigned to a DateTime instance and not a string representation of a Date
            sc.Parameters.Add(new SqlParameter("@dateOfBirth", SqlDbType.Date){ Value =
newEmployee.DateOfBirth });

            // best practice - open your connection as late as possible unless you need to
verify that the database connection is valid and wont fail and the proceeding code execution
takes a long time (not the case here)
            con.Open();
            sc.ExecuteNonQuery();
        }

        // the end of the using block will close and dispose the SqlConnection
        // best practice - end the using block as soon as possible to release the database
connection
    }
}

// supporting class used as parameter for example
public class Employee
```

```
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
}
```

Le migliori pratiche per lavorare con ADO.NET

- La regola generale è di aprire la connessione per un tempo minimo. Chiudere esplicitamente la connessione al termine dell'esecuzione della procedura per ripristinare l'oggetto di connessione sul pool di connessioni. Dimensione massima pool di connessione predefinita = 100. Poiché il pool di connessioni migliora le prestazioni della connessione fisica a SQL Server. Pool di connessioni in SQL Server
- Avvolgi tutte le connessioni del database in un blocco utilizzando in modo che siano sempre chiuse e disposte anche nel caso di un'eccezione. Vedi usando Statement (C # Reference) per ulteriori informazioni sull'uso di istruzioni
- Recupera le stringhe di connessione per nome da app.config o web.config (a seconda del tipo di applicazione)
    - Ciò richiede un riferimento all'assembly a `System.configuration`
    - Vedere Stringhe di connessione e file di configurazione per ulteriori informazioni su come strutturare il file di configurazione
- Usa sempre i parametri per i valori in entrata a
    - Evita gli attacchi di iniezione sql
    - Evitare errori se si utilizza un testo non valido come includere una virgoletta singola che è l'equivalente in sql dell'escape o dell'avvio di una stringa (varchar / nvarchar)
    - Lasciando che il provider di database riutilizzi i piani di query (non supportati da tutti i provider di database) aumenta l'efficienza
- Quando si lavora con i parametri
    - Il tipo di parametri Sql e la mancata corrispondenza delle dimensioni è una causa comune di errore inserimento / aggiornamento / selezione
    - Dai ai tuoi parametri Sql nomi significativi proprio come fai con le variabili nel tuo codice
    - Specificare il tipo di dati del database della colonna che si sta utilizzando, questo garantisce che non vengano utilizzati tipi di parametri errati che potrebbero portare a risultati imprevisti
    - Convalida i tuoi parametri in entrata prima di passarli al comando (come dice il proverbio, " garbage in, garbage out "). Convalida i valori in entrata il prima possibile nello stack
    - Utilizzare i tipi corretti durante l'assegnazione dei valori dei parametri, ad esempio: non assegnare il valore stringa di un DateTime, invece assegnare un'istanza DateTime effettiva al valore del parametro
    - Specificare la dimensione dei parametri del tipo di stringa. Ciò è dovuto al fatto che SQL Server può riutilizzare i piani di esecuzione se i parametri corrispondono nel tipo *e nella* dimensione. Usa -1 per MAX
    - Non utilizzare il metodo AddWithValue , il motivo principale è che è molto facile dimenticare di specificare il tipo di parametro o la precisione / la scala quando

necessario. Per ulteriori informazioni vedi Come possiamo smettere di usare AddWithValue già?

- Quando si utilizzano connessioni di database
  - Apri la connessione il più tardi possibile e chiudila il prima possibile. Questa è una linea guida generale quando si lavora con qualsiasi risorsa esterna
  - Non condividere mai istanze di connessione al database (esempio: avere un host singleton un'istanza condivisa di tipo `SqlConnection`). Fai in modo che il codice crei sempre una nuova istanza di connessione al database quando è necessario e quindi richiedi al codice chiamante di eliminarlo e "buttalo via" quando è terminato. La ragione di questo è
    - La maggior parte dei provider di database ha una sorta di pool di connessioni, quindi la creazione di nuove connessioni gestite è economica
    - Elimina eventuali errori futuri se il codice inizia a lavorare con più thread

Utilizzo di interfacce comuni per astrarre le classi specifiche del fornitore

```
var providerName = "System.Data.SqlClient";    //Oracle.ManagedDataAccess.Client, IBM.Data.DB2
var connectionString = "{your-connection-string}";
//you will probably get the above two values in the ConnectionStringSettings object from
.config file

var factory = DbProviderFactories.GetFactory(providerName);
using(var connection = factory.CreateConnection()) {    //IDbConnection
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = connection.CreateCommand()) {    //IDbCommand
        command.CommandText = "{query}";

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

Leggi ADO.NET online: https://riptutorial.com/it/dot-net/topic/3589/ado-net

Capitolo 4: Alberi di espressione

Osservazioni

Gli alberi di espressione sono strutture di dati utilizzate per rappresentare le espressioni di codice in .NET Framework. Possono essere generati dal codice e attraversati a livello di codice per tradurre il codice in un'altra lingua o eseguirlo. Il generatore più popolare di Expression Trees è il compilatore C # stesso. Il compilatore C # può generare alberi di espressioni se viene assegnata un'espressione lambda a una variabile di tipo Expression <Func <... >>. Di solito questo accade nel contesto di LINQ. Il consumatore più popolare è il provider LINQ di Entity Framework. Consuma gli alberi di espressioni dati a Entity Framework e genera un codice SQL equivalente che viene quindi eseguito sul database.

Examples

Albero delle espressioni semplici generato dal compilatore C

Considera il seguente codice C #

```
Expression<Func<int, int>> expression = a => a + 1;
```

Poiché il compilatore C # rileva che l'espressione lambda è assegnata a un tipo di espressione anziché a un tipo delegato, genera un albero di espressioni approssimativamente equivalente a questo codice

```
ParameterExpression parameterA = Expression.Parameter(typeof(int), "a");
var expression = (Expression<Func<int, int>>)Expression.Lambda(
                                                Expression.Add(
                                                    parameterA,
                                                    Expression.Constant(1)),
                                                parameterA);
```

La radice dell'albero è l'espressione lambda che contiene un corpo e un elenco di parametri. Il lambda ha 1 parametro chiamato "a". Il corpo è una singola espressione di tipo CLR BinaryExpression e NodeType di Aggiungi. Questa espressione rappresenta l'aggiunta. Ha due sottoespressioni denotate come sinistra e destra. A sinistra è ParameterExpression per il parametro "a" e Right è una espressione costante con il valore 1.

L'uso più semplice di questa espressione è la stampa:

```
Console.WriteLine(expression); //prints a => (a + 1)
```

Che stampa il codice C # equivalente.

L'albero delle espressioni può essere compilato in un delegato C # ed eseguito dal CLR

```
Func<int, int> lambda = expression.Compile();
Console.WriteLine(lambda(2)); //prints 3
```

Solitamente le espressioni sono tradotte in altri linguaggi come SQL, ma possono anche essere utilizzate per invocare membri privati, protetti e interni di tipi pubblici o non pubblici come alternativa a Reflection.

creazione di un predicato del campo modulo == valore

Per creare un'espressione come `_ => _.Field == "VALUE"` in fase di runtime.

Dato un predicato `_ => _.Field` e un valore stringa `"VALUE"` , crea un'espressione che verifica se il predicato è vero o meno.

L'espressione è adatta per:

- `IQueryable<T>` , `IEnumerable<T>` per testare il predicato.
- Entity Framework o `Linq` to `SQL` per creare una clausola `Where` che verifica il predicato.

Questo metodo creerà un'espressione `Equal` appropriata che verifica se `Field` equivale a `"VALUE"` .

```
public static Expression<Func<T, bool>> BuildEqualPredicate<T>(
    Expression<Func<T, string>> memberAccessor,
    string term)
{
    var toString = Expression.Convert(Expression.Constant(term), typeof(string));
    Expression expression = Expression.Equal(memberAccessor.Body, toString);
    var predicate = Expression.Lambda<Func<T, bool>>(
        expression,
        memberAccessor.Parameters);
    return predicate;
}
```

Il predicato può essere utilizzato includendo il predicato in un metodo di estensione `Where` .

```
var predicate = PredicateExtensions.BuildEqualPredicate<Entity>(
    _ => _.Field,
    "VALUE");
var results = context.Entity.Where(predicate).ToList();
```

Espressione per il recupero di un campo statico

Avere un esempio digita in questo modo:

```
public TestClass
{
    public static string StaticPublicField = "StaticPublicFieldValue";
}
```

Possiamo recuperare il valore di StaticPublicField:

```
var fieldExpr = Expression.Field(null, typeof(TestClass), "StaticPublicField");
var labmda = Expression.Lambda<Func<string>>(fieldExpr);
```

Può quindi essere compilato in un delegato per il recupero del valore del campo.

```
Func<string> retriever = lambda.Compile();
var fieldValue = retriever();
```

// Il risultato fieldValue è StaticPublicFieldValue

InvocationExpression Class

La classe InvocationExpression consente il richiamo di altre espressioni lambda che fanno parte dello stesso albero di espressione.

Li creo con il metodo `Expression.Invoke` statico.

Problema Vogliamo ottenere gli articoli che hanno "auto" nella descrizione. Dobbiamo controllarlo per null prima di cercare una stringa all'interno, ma non vogliamo che venga chiamata eccessivamente, in quanto il calcolo potrebbe essere costoso.

```
using System;
using System.Linq;
using System.Linq.Expressions;

public class Program
{
    public static void Main()
    {
        var elements = new[] {
            new Element { Description = "car" },
            new Element { Description = "cargo" },
            new Element { Description = "wheel" },
            new Element { Description = null },
            new Element { Description = "Madagascar" },
        };

        var elementIsInterestingExpression = CreateSearchPredicate(
            searchTerm: "car",
            whereToSearch: (Element e) => e.Description);

        Console.WriteLine(elementIsInterestingExpression.ToString());

        var elementIsInteresting = elementIsInterestingExpression.Compile();
        var interestingElements = elements.Where(elementIsInteresting);
        foreach (var e in interestingElements)
        {
            Console.WriteLine(e.Description);
        }

        var countExpensiveComputations = 0;
        Action incCount = () => countExpensiveComputations++;
        elements
            .Where(
                CreateSearchPredicate(
                    "car",
```

```
                    (Element e) => ExpensivelyComputed(
                        e, incCount
                    )
                ).Compile()
            )
            .Count();

        Console.WriteLine("Property extractor is called {0} times.",
countExpensiveComputations);
    }

    private class Element
    {
        public string Description { get; set; }
    }

    private static string ExpensivelyComputed(Element source, Action count)
    {
        count();
        return source.Description;
    }

    private static Expression<Func<T, bool>> CreateSearchPredicate<T>(
            string searchTerm,
            Expression<Func<T, string>> whereToSearch)
    {
        var extracted = Expression.Parameter(typeof(string), "extracted");

        Expression<Func<string, bool>> coalesceNullCheckWithSearch =
            Expression.Lambda<Func<string, bool>>(
                Expression.AndAlso(
                    Expression.Not(
                        Expression.Call(typeof(string), "IsNullOrEmpty", null, extracted)
                    ),
                    Expression.Call(extracted, "Contains", null,
Expression.Constant(searchTerm))
                ),
                extracted);

        var elementParameter = Expression.Parameter(typeof(T), "element");

        return Expression.Lambda<Func<T, bool>>(
            Expression.Invoke(
                coalesceNullCheckWithSearch,
                Expression.Invoke(whereToSearch, elementParameter)
            ),
            elementParameter
        );
    }
}
```

Produzione

```
element => Invoke(extracted => (Not(IsNullOrEmpty(extracted)) AndAlso
extracted.Contains("car")), Invoke(e => e.Description, element))
car
cargo
Madagascar
Predicate is called 5 times.
```

La prima cosa da notare è come l'accesso vero e proprio, racchiuso in un richiamo:

```
Invoke(e => e.Description, element)
```

, e questa è l'unica parte che tocca `e.Description` e al suo posto, il parametro `extracted` di tipo `string` viene passato a quello successivo:

```
(Not(IsNullOrEmpty(extracted)) AndAlso extracted.Contains("car"))
```

Un'altra cosa importante da notare qui è `AndAlso` . Calcola solo la parte sinistra, se la prima parte restituisce 'falso'. È un errore comune utilizzare l'operatore bit per bit "And" al posto di esso, che calcola sempre entrambe le parti e non riuscirebbe con una NullReferenceException in questo esempio.

Leggi Alberi di espressione online: https://riptutorial.com/it/dot-net/topic/2657/alberi-di-espressione

Capitolo 5: Carica file e dati POST sul server web

Examples

Carica il file con WebRequest

Per inviare un file e formattare i dati in una singola richiesta, il contenuto dovrebbe avere il tipo multipart / form-data .

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Threading.Tasks;

public async Task<string> UploadFile(string url, string filename,
    Dictionary<string, object> postData)
{
    var request = WebRequest.CreateHttp(url);
    var boundary = $"{Guid.NewGuid():N}"; // boundary will separate each parameter
    request.ContentType = $"multipart/form-data; {nameof(boundary)}={boundary}";
    request.Method = "POST";

    using (var requestStream = request.GetRequestStream())
    using (var writer = new StreamWriter(requestStream))
    {
        foreach (var data in postData)
            await writer.WriteAsync( // put all POST data into request
                $"\r\n--{boundary}\r\nContent-Disposition: " +
                $"form-data; name=\"{data.Key}\"\r\n\r\n{data.Value}");

        await writer.WriteAsync( // file header
            $"\r\n--{boundary}\r\nContent-Disposition: " +
            $"form-data; name=\"File\"; filename=\"{Path.GetFileName(filename)}\"\r\n" +
            "Content-Type: application/octet-stream\r\n\r\n");

        await writer.FlushAsync();
        using (var fileStream = File.OpenRead(filename))
            await fileStream.CopyToAsync(requestStream);

        await writer.WriteAsync($"\r\n--{boundary}--\r\n");
    }

    using (var response = (HttpWebResponse) await request.GetResponseAsync())
    using (var responseStream = response.GetResponseStream())
    {
        if (responseStream == null)
            return string.Empty;
        using (var reader = new StreamReader(responseStream))
            return await reader.ReadToEndAsync();
    }
}
```

Uso:

```
var response = await uploader.UploadFile("< YOUR URL >", "< PATH TO YOUR FILE >",
    new Dictionary<string, object>
    {
        {"Comment", "test"},
        {"Modified", DateTime.Now }
    });
```

Leggi Carica file e dati POST sul server web online: https://riptutorial.com/it/dot-net/topic/10845/carica-file-e-dati-post-sul-server-web

Capitolo 6: Client HTTP

Osservazioni

Le RFC HTTP / 1.1 attualmente rilevanti sono:

- 7230: Sintassi dei messaggi e routing
- 7231: semantica e contenuto
- 7232: richieste condizionali
- 7233: Richieste di intervallo
- 7234: memorizzazione nella cache
- 7235: Authenticaion
- 7239: Estensione HTTP inoltrata
- 7240: Preferisci intestazione per HTTP

Esistono anche le seguenti RFC informative:

- 7236: registrazioni degli schemi di autenticazione
- 7237: registrazioni dei metodi

E la RFC sperimentale:

- 7238: Il codice di stato del protocollo di trasferimento ipertestuale 308 (reindirizzamento permanente)

Protocolli correlati:

- 4918: estensioni HTTP per Web Distributed Authoring e Versioning (WebDAV)
- 4791: Calendaring Extensions su WebDAV (CalDAV)

Examples

Leggere la risposta GET come stringa usando System.Net.HttpWebRequest

```
string requestUri = "http://www.example.com";
string responseData;

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(parameters.Uri);
WebResponse response = request.GetResponse();

using (StreamReader responseReader = new StreamReader(response.GetResponseStream()))
{
    responseData = responseReader.ReadToEnd();
}
```

Leggere la risposta GET come stringa utilizzando System.Net.WebClient

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new WebClient())
{
    responseData = client.DownloadString(requestUri);
}
```

Leggere la risposta GET come stringa utilizzando System.Net.HttpClient

`HttpClient` è disponibile tramite NuGet: Librerie client Microsoft HTTP .

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new HttpClient())
{
    using(var response = client.GetAsync(requestUri).Result)
    {
       response.EnsureSuccessStatusCode();
       responseData = response.Content.ReadAsStringAsync().Result;
    }
}
```

Invio di una richiesta POST con un payload di stringa utilizzando System.Net.HttpWebRequest

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri)
{
  Method = requestMethod,
  ContentType = contentType,
};

byte[] bytes = Encoding.UTF8.GetBytes(requestBodyString);
Stream stream = request.GetRequestStream();
stream.Write(bytes, 0, bytes.Length);
stream.Close();

HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

Invio di una richiesta POST con un payload di stringa utilizzando System.Net.WebClient

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

byte[] responseBody;
```

```
byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);

using (var client = new WebClient())
{
    client.Headers[HttpRequestHeader.ContentType] = contentType;
    responseBody = client.UploadData(requestUri, requestMethod, requestBodyBytes);
}
```

Invio di una richiesta POST con un payload di stringa utilizzando System.Net.HttpClient

`HttpClient` è disponibile tramite NuGet: Librerie client Microsoft HTTP .

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

var request = new HttpRequestMessage
{
    RequestUri = requestUri,
    Method = requestMethod,
};

byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);
request.Content = new ByteArrayContent(requestBodyBytes);

request.Content.Headers.ContentType = new MediaTypeHeaderValue(contentType);

HttpResponseMessage result = client.SendAsync(request).Result;
result.EnsureSuccessStatusCode();
```

Downloader di base HTTP che utilizza System.Net.Http.HttpClient

```
using System;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

class HttpGet
{
    private static async Task DownloadAsync(string fromUrl, string toFile)
    {
        using (var fileStream = File.OpenWrite(toFile))
        {
            using (var httpClient = new HttpClient())
            {
                Console.WriteLine("Connecting...");
                using (var networkStream = await httpClient.GetStreamAsync(fromUrl))
                {
                    Console.WriteLine("Downloading...");
                    await networkStream.CopyToAsync(fileStream);
                    await fileStream.FlushAsync();
                }
            }
        }
```

```
    }

    static void Main(string[] args)
    {
        try
        {
            Run(args).Wait();
        }
        catch (Exception ex)
        {
            if (ex is AggregateException)
                ex = ((AggregateException)ex).Flatten().InnerExceptions.First();

            Console.WriteLine("--- Error: " +
                (ex.InnerException?.Message ?? ex.Message));
        }
    }
    static async Task Run(string[] args)
    {
        if (args.Length < 2)
        {
            Console.WriteLine("Basic HTTP downloader");
            Console.WriteLine();
            Console.WriteLine("Usage: httpget <url>[<:port>] <file>");
            return;
        }

        await DownloadAsync(fromUrl: args[0], toFile: args[1]);

        Console.WriteLine("Done!");
    }
}
```

Leggi Client HTTP online: https://riptutorial.com/it/dot-net/topic/32/client-http

Capitolo 7: CLR

Examples

Un'introduzione a Common Language Runtime

Common Language Runtime (CLR) è un ambiente di macchina virtuale e parte di .NET Framework. Contiene:

- Un linguaggio bytecode portatile chiamato **Common Intermediate Language** (abbreviato CIL o IL)
- Un compilatore Just-In-Time che genera codice macchina
- Un garbage collector di tracciamento che fornisce la gestione automatica della memoria
- Supporto per processi secondari leggeri denominati AppDomain
- Meccanismi di sicurezza attraverso i concetti di codice verificabile e livelli di fiducia

Il codice che viene eseguito nel CLR viene definito *codice gestito* per distinguerlo dal codice in esecuzione all'esterno del CLR (in genere codice nativo) che viene indicato come *codice non gestito* . Esistono vari meccanismi che facilitano l'interoperabilità tra codice gestito e non gestito.

Leggi CLR online: https://riptutorial.com/it/dot-net/topic/3942/clr

Capitolo 8: collezioni

Osservazioni

Esistono diversi tipi di raccolta:

- `Array`
- `List`
- `Queue`
- `SortedList`
- `Stack`
- Dizionario

Examples

Creazione di un elenco inizializzato con tipi personalizzati

```
public class Model
{
    public string Name { get; set; }
    public bool? Selected { get; set; }
}
```

Qui abbiamo una classe senza costruttore con due proprietà: `Name` e una proprietà booleana nullable `Selected` . Se volessimo inizializzare un `List<Model>` , ci sono diversi modi per eseguirlo.

```
var SelectedEmployees = new List<Model>
 {
      new Model() {Name = "Item1", Selected = true},
      new Model() {Name = "Item2", Selected = false},
      new Model() {Name = "Item3", Selected = false},
      new Model() {Name = "Item4"}
 };
```

Qui, stiamo creando diverse `new` istanze della nostra classe `Model` e inizializzandole con i dati. Cosa succede se abbiamo aggiunto un costruttore?

```
public class Model
{

    public Model(string name, bool? selected = false)
    {
        Name = name;
        selected = Selected;
    }
    public string Name { get; set; }
    public bool? Selected { get; set; }
}
```

Questo ci consente di inizializzare la nostra lista in modo *leggermente* diverso.

```
var SelectedEmployees = new List<Model>
{
    new Model("Mark", true),
    new Model("Alexis"),
    new Model("")
};
```

Che dire di una classe in cui una delle proprietà è una classe stessa?

```
public class Model
{
    public string Name { get; set; }
    public bool? Selected { get; set; }
}

public class ExtendedModel : Model
{
    public ExtendedModel()
    {
        BaseModel = new Model();
    }

    public Model BaseModel { get; set; }
    public DateTime BirthDate { get; set; }
}
```

Si noti che abbiamo ripristinato il costruttore della classe `Model` per semplificare un po 'l'esempio.

```
var SelectedWithBirthDate = new List<ExtendedModel>
{
    new ExtendedModel()
    {
        BaseModel = new Model { Name = "Mark", Selected = true},
        BirthDate = new DateTime(2015, 11, 23)
    },
                    new ExtendedModel()
    {
        BaseModel = new Model { Name = "Random"},
        BirthDate = new DateTime(2015, 11, 23)
    }
};
```

Nota che possiamo scambiare la nostra `List<ExtendedModel>` con `Collection<ExtendedModel>` , `ExtendedModel[]` , `object[]` , o anche semplicemente `[]` .

Coda

C'è una collezione in .Net usata per gestire i valori in una `Queue` che usa il concetto FIFO (first-in first-out) . Le basi delle code è il metodo `Enqueue(T item)` che viene utilizzato per aggiungere elementi nella coda e `Dequeue()` che viene utilizzato per ottenere il primo elemento e rimuoverlo dalla coda. La versione generica può essere utilizzata come il seguente codice per una coda di stringhe.

Innanzitutto, aggiungi lo spazio dei nomi:

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

e usalo:

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("John");
queue.Enqueue("Paul");
queue.Enqueue("George");
queue.Enqueue("Ringo");

string dequeueValue;
dequeueValue = queue.Dequeue(); // return John
dequeueValue = queue.Dequeue(); // return Paul
dequeueValue = queue.Dequeue(); // return George
dequeueValue = queue.Dequeue(); // return Ringo
```

Esiste una versione non generica del tipo, che funziona con gli oggetti.

Lo spazio dei nomi è:

```
using System.Collections;
```

Ad un esempio di codice per coda non generica:

```
Queue queue = new Queue();
queue.Enqueue("Hello World"); // string
queue.Enqueue(5); // int
queue.Enqueue(1d); // double
queue.Enqueue(true); // bool
queue.Enqueue(new Product()); // Product object

object dequeueValue;
dequeueValue = queue.Dequeue(); // return Hello World (string)
dequeueValue = queue.Dequeue(); // return 5 (int)
dequeueValue = queue.Dequeue(); // return 1d (double)
dequeueValue = queue.Dequeue(); // return true (bool)
dequeueValue = queue.Dequeue(); // return Product (Product type)
```

Esiste anche un metodo chiamato Peek () che restituisce l'oggetto all'inizio della coda senza rimuoverlo.

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(10);
queue.Enqueue(20);
queue.Enqueue(30);
queue.Enqueue(40);
queue.Enqueue(50);

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

L'output (senza rimuovere):

```
10
20
30
40
50
```

Pila

C'è una collezione in .Net usata per gestire i valori in uno `Stack` che usa il concetto LIFO (last-in first-out) . Le basi degli stack sono il metodo `Push(T item)` che viene utilizzato per aggiungere elementi nello stack e `Pop()` che viene utilizzato per ottenere l'ultimo elemento aggiunto e rimuoverlo dallo stack. La versione generica può essere utilizzata come il seguente codice per una coda di stringhe.

Innanzitutto, aggiungi lo spazio dei nomi:

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

e usalo:

```
Stack<string> stack = new Stack<string>();
stack.Push("John");
stack.Push("Paul");
stack.Push("George");
stack.Push("Ringo");

string value;
value = stack.Pop(); // return Ringo
value = stack.Pop(); // return George
value = stack.Pop(); // return Paul
value = stack.Pop(); // return John
```

Esiste una versione non generica del tipo, che funziona con gli oggetti.

Lo spazio dei nomi è:

```
using System.Collections;
```

E un esempio di codice di stack non generico:

```
Stack stack = new Stack();
stack.Push("Hello World"); // string
stack.Push(5); // int
stack.Push(1d); // double
stack.Push(true); // bool
stack.Push(new Product()); // Product object

object value;
value = stack.Pop(); // return Product (Product type)
value = stack.Pop(); // return true (bool)
value = stack.Pop(); // return 1d (double)
value = stack.Pop(); // return 5 (int)
value = stack.Pop(); // return Hello World (string)
```

Esiste anche un metodo chiamato Peek () che restituisce l'ultimo elemento aggiunto ma senza rimuoverlo dallo `Stack` .

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);

var lastValueAdded = stack.Peek(); // 20
```

È possibile iterare sugli elementi in pila e rispetterà l'ordine dello stack (LIFO).

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);
stack.Push(30);
stack.Push(40);
stack.Push(50);

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

L'output (senza rimuovere):

```
50
40
30
20
10
```

Utilizzo degli inizializzatori di raccolta

Alcuni tipi di raccolta possono essere inizializzati al momento della dichiarazione. Ad esempio, la seguente istruzione crea e inizializza i `numbers` con alcuni numeri interi:

```
List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

Internamente, il compilatore C # converte effettivamente questa inizializzazione in una serie di chiamate al metodo Add. Di conseguenza, è possibile utilizzare questa sintassi solo per le raccolte che supportano effettivamente il metodo `Add` .

> Le classi `Stack<T>` e `Queue<T>` non la supportano.

Per raccolte complesse come la classe `Dictionary<TKey, TValue>` , che `Dictionary<TKey, TValue>` coppie chiave / valore, è possibile specificare ogni coppia chiave / valore come un tipo anonimo nell'elenco di inizializzazione.

```
Dictionary<int, string> employee = new Dictionary<int, string>()
     {{44, "John"}, {45, "Bob"}, {47, "James"}, {48, "Franklin"}};
```

Il primo elemento di ciascuna coppia è la chiave e il secondo è il valore.

Leggi collezioni online: https://riptutorial.com/it/dot-net/topic/30/collezioni

Capitolo 9: Compilatore JIT

introduzione

La compilazione JIT, o compilazione just-in-time, è un approccio alternativo all'interpretazione del codice o alla compilazione in anticipo. La compilazione JIT è utilizzata nel framework .NET. Il codice CLR (C #, F #, Visual Basic, ecc.) Viene prima compilato in qualcosa chiamato Interpreted Language, o IL. Questo è un codice di livello inferiore che è più vicino al codice macchina, ma non è specifico per la piattaforma. Piuttosto, in fase di esecuzione, questo codice è compilato in codice macchina per il sistema pertinente.

Osservazioni

Perché usare la compilazione JIT?

- Migliore compatibilità: ogni linguaggio CLR ha bisogno di un solo compilatore per IL, e questo IL può essere eseguito su qualsiasi piattaforma su cui può essere convertito in codice macchina.
- Velocità: la compilazione JIT tenta di combinare la velocità di esecuzione del codice compilato in anticipo e la flessibilità dell'interpretazione (può analizzare il codice che verrà eseguito per potenziali ottimizzazioni prima della compilazione)

Pagina di Wikipedia per ulteriori informazioni sulla compilazione JIT in generale:
https://en.wikipedia.org/wiki/Just-in-time_compilation

Examples

Esempio di compilazione IL

Semplice applicazione Hello World:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

Codice IL equivalente (che verrà compilato JIT)

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 4.6.1055.0
```

```
//  Copyright (c) Microsoft Corporation.  All rights reserved.



// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
  .ver 4:0:0:0
}
.assembly HelloWorld
{
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01
00 08 00 00 00 00 00 )
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() = ( 01 00 01
00 54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx

63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.

  // --- The following custom attribute is added automatically, do not uncomment -------
  //  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute::.ctor(valuetype
[mscorlib]System.Diagnostics.DebuggableAttribute/DebuggingModes) = ( 01 00 07 01 00 00 00 00 )


  .custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string) = (
01 00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
  .custom instance void
[mscorlib]System.Reflection.AssemblyDescriptionAttribute::.ctor(string) = ( 01 00 00 00 00 )
  .custom instance void
[mscorlib]System.Reflection.AssemblyConfigurationAttribute::.ctor(string) = ( 01 00 00 00 00 )

  .custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(string) =
( 01 00 00 00 00 )
  .custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::.ctor(string) =
( 01 00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
  .custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string)
= ( 01 00 12 43 6F 70 79 72 69 67 68 74 20 C2 A9 20   // ...Copyright ..

20 32 30 31 37 00 00 )                            //  2017..
  .custom instance void [mscorlib]System.Reflection.AssemblyTrademarkAttribute::.ctor(string)
= ( 01 00 00 00 00 )
  .custom instance void
[mscorlib]System.Runtime.InteropServices.ComVisibleAttribute::.ctor(bool) = ( 01 00 00 00 00 )

  .custom instance void [mscorlib]System.Runtime.InteropServices.GuidAttribute::.ctor(string)
= ( 01 00 24 33 30 38 62 33 64 38 36 2D 34 31 37 32   // ..$308b3d86-4172

2D 34 30 32 32 2D 61 66 63 63 2D 33 66 38 65 33   // -4022-afcc-3f8e3

32 33 33 63 35 62 30 00 00 )                      // 233c5b0..
  .custom instance void
[mscorlib]System.Reflection.AssemblyFileVersionAttribute::.ctor(string) = ( 01 00 07 31 2E 30
2E 30 2E 30 00 00 )             // ...1.0.0.0..
  .custom instance void
[mscorlib]System.Runtime.Versioning.TargetFrameworkAttribute::.ctor(string) = ( 01 00 1C 2E 4E
45 54 46 72 61 6D 65 77 6F 72 6B   // ....NETFramework

2C 56 65 72 73 69 6F 6E 3D 76 34 2E 35 2E 32 01   // ,Version=v4.5.2.
```

```
00 54 0E 14 46 72 61 6D 65 77 6F 72 6B 44 69 73   // .T..FrameworkDis

70 6C 61 79 4E 61 6D 65 14 2E 4E 45 54 20 46 72   // playName..NET Fr

61 6D 65 77 6F 72 6B 20 34 2E 35 2E 32 )          // amework 4.5.2
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module HelloWorld.exe
// MVID: {2A7E1D59-1272-4B47-85F6-D7E1ED057831}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00020003    //  ILONLY 32BITPREFERRED
// Image base: 0x0000021C70230000


// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit HelloWorld.Program
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       13 (0xd)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello World"
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  } // end of method Program::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       8 (0x8)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  nop
    IL_0007:  ret
  } // end of method Program::.ctor

} // end of class HelloWorld.Program
```

Generato con MS ILDASM tool (IL disassembler)

Leggi Compilatore JIT online: https://riptutorial.com/it/dot-net/topic/9222/compilatore-jit

Capitolo 10: Contesti di sincronizzazione

Osservazioni

Un contesto di sincronizzazione è un'astrazione che consente di utilizzare il codice per passare unità di lavoro a uno scheduler, senza richiedere la consapevolezza di come verrà pianificato il lavoro.

I contesti di sincronizzazione vengono tradizionalmente utilizzati per garantire che il codice venga eseguito su un thread specifico. Nelle applicazioni WPF e Winforms, il framework di presentazione fornisce un `SynchronizationContext` rappresenta il thread dell'interfaccia utente. In questo modo, `SynchronizationContext` può essere considerato un modello produttore-consumatore per i delegati. Un thread di lavoro *produrrà* il codice eseguibile (il delegato) e lo accoderà o *consumerà* dal loop di messaggi dell'interfaccia utente.

La Libreria parallela attività fornisce funzionalità per l'acquisizione e l'utilizzo automatico dei contesti di sincronizzazione.

Examples

Esegui codice sul thread dell'interfaccia utente dopo aver eseguito il lavoro in background

Questo esempio mostra come aggiornare un componente dell'interfaccia utente da un thread in background utilizzando un `SynchronizationContext`

```
void Button_Click(object sender, EventArgs args)
{
    SynchronizationContext context = SynchronizationContext.Current;
    Task.Run(() =>
    {
        for(int i = 0; i < 10; i++)
        {
            Thread.Sleep(500); //simulate work being done
            context.Post(ShowProgress, "Work complete on item " + i);
        }
    }
}

void UpdateCallback(object state)
{
    // UI can be safely updated as this method is only called from the UI thread
    this.MyTextBox.Text = state as string;
}
```

In questo esempio, se si è tentato di aggiornare direttamente `MyTextBox.Text` all'interno del ciclo `for` , si otterrebbe un errore di threading. `UpdateCallback` azione `UpdateCallback` su `SynchronizationContext` , la casella di testo viene aggiornata sullo stesso thread del resto dell'interfaccia utente.

In pratica, gli aggiornamenti dei progressi dovrebbero essere eseguiti utilizzando un'istanza di `System.IProgress<T>` . L'implementazione predefinita `System.Progress<T>` acquisisce automaticamente il contesto di sincronizzazione su cui è stato creato.

Leggi Contesti di sincronizzazione online: https://riptutorial.com/it/dot-net/topic/5407/contesti-di-sincronizzazione

Capitolo 11: Contratti di codice

Osservazioni

I contratti di codice consentono l'analisi compile o runtime delle condizioni pre / post dei metodi e delle condizioni invarianti per gli oggetti. Queste condizioni possono essere utilizzate per garantire che i chiamanti e il valore restituito soddisfino gli stati validi per l'elaborazione dell'applicazione. Altri usi per i contratti di codice includono la generazione di documentazione.

Examples

presupposti

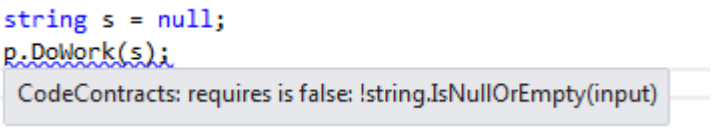
Le precondizioni consentono ai metodi di fornire i valori minimi richiesti per i parametri di input

Esempio...

```
void DoWork(string input)
{
    Contract.Requires(!string.IsNullOrEmpty(input));

    //do work
}
```

Risultato dell'analisi statica ...



postconditions

Le post-condizioni assicurano che i risultati restituiti da un metodo corrispondano alla definizione fornita. Ciò fornisce al chiamante una definizione del risultato atteso. Le post-condizioni possono essere consentite per le implosazioni semplificate poiché alcuni risultati possibili possono essere forniti dall'analizzatore statico.

Esempio...

```
string GetValue()
{
    Contract.Ensures(Contract.Result<string>() != null);

    return null;
}
```

Risultato dell'analisi statica ...

```
string GetValue()
{
    Contract.Ensures(Contract.Result<string>() != null);

    return null;
}
```
CodeContracts: Invoking method 'GetValue' will always lead to an error. If this is wanted, consider adding Contract.Requires(false) to

Contratti per interfacce

Utilizzando i Contratti di codice è possibile applicare un contratto a un'interfaccia. Questo viene fatto dichiarando una classe astratta che implora le interfacce. L'interfaccia deve essere contrassegnata con `ContractClassAttribute` e la definizione del contratto (la classe astratta) deve essere contrassegnata con `ContractClassForAttribute`

C # Esempio ...

```
[ContractClass(typeof(MyInterfaceContract))]
public interface IMyInterface
{
    string DoWork(string input);
}
//Never inherit from this contract defintion class
[ContractClassFor(typeof(IMyInterface))]
internal abstract class MyInterfaceContract : IMyInterface
{
    private MyInterfaceContract() { }

    public string DoWork(string input)
    {
        Contract.Requires(!string.IsNullOrEmpty(input));
        Contract.Ensures(!string.IsNullOrEmpty(Contract.Result<string>()));
        throw new NotSupportedException();
    }
}
public class MyInterfaceImplmentation : IMyInterface
{
    public string DoWork(string input)
    {
        return input;
    }
}
```

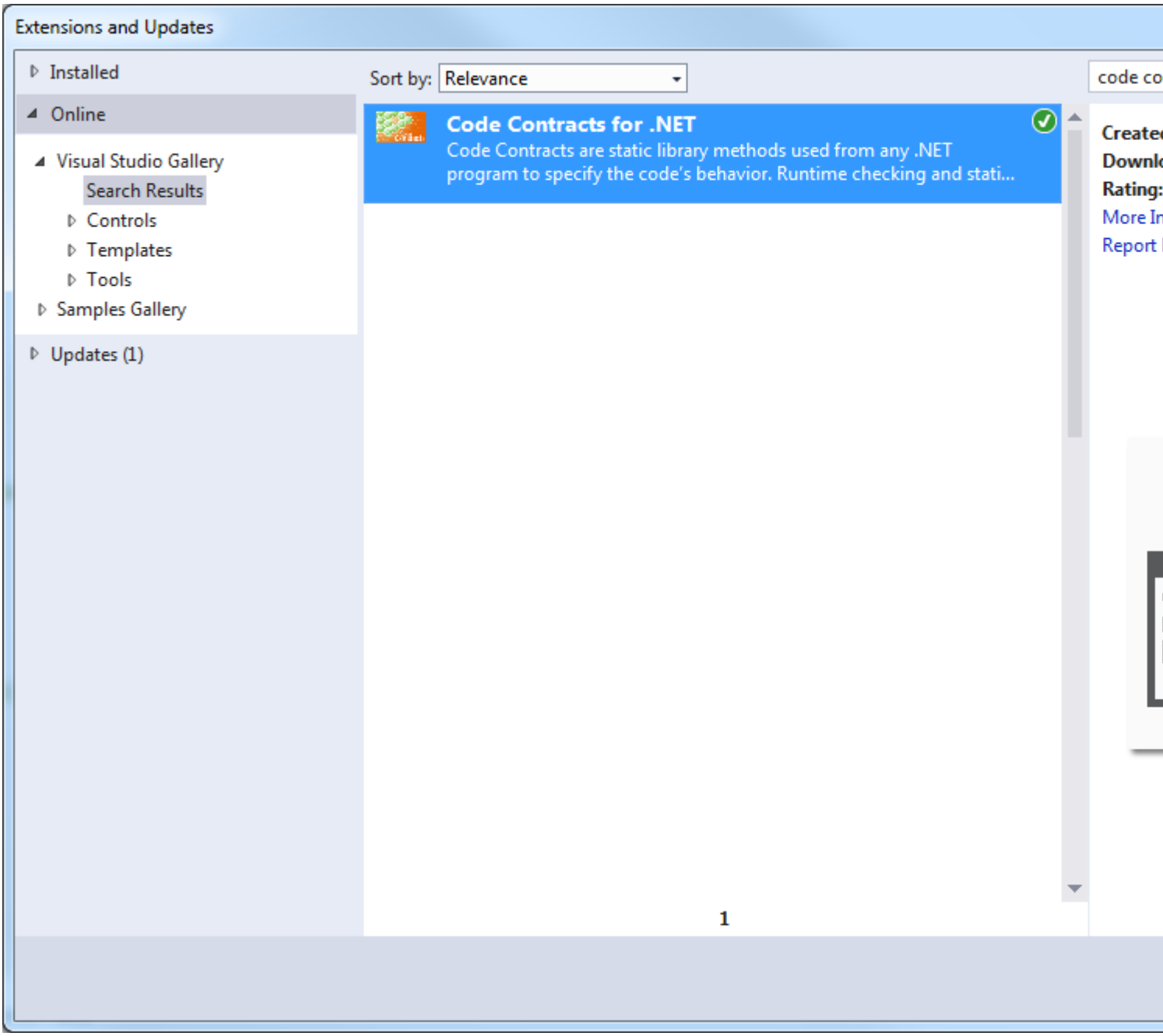
Risultato dell'analisi statica ...

```
var m = new MyInterfaceImplmentation();
var ret = m.DoWork(null);
```
CodeContracts: requires is false: !string.IsNullOrEmpty(input)

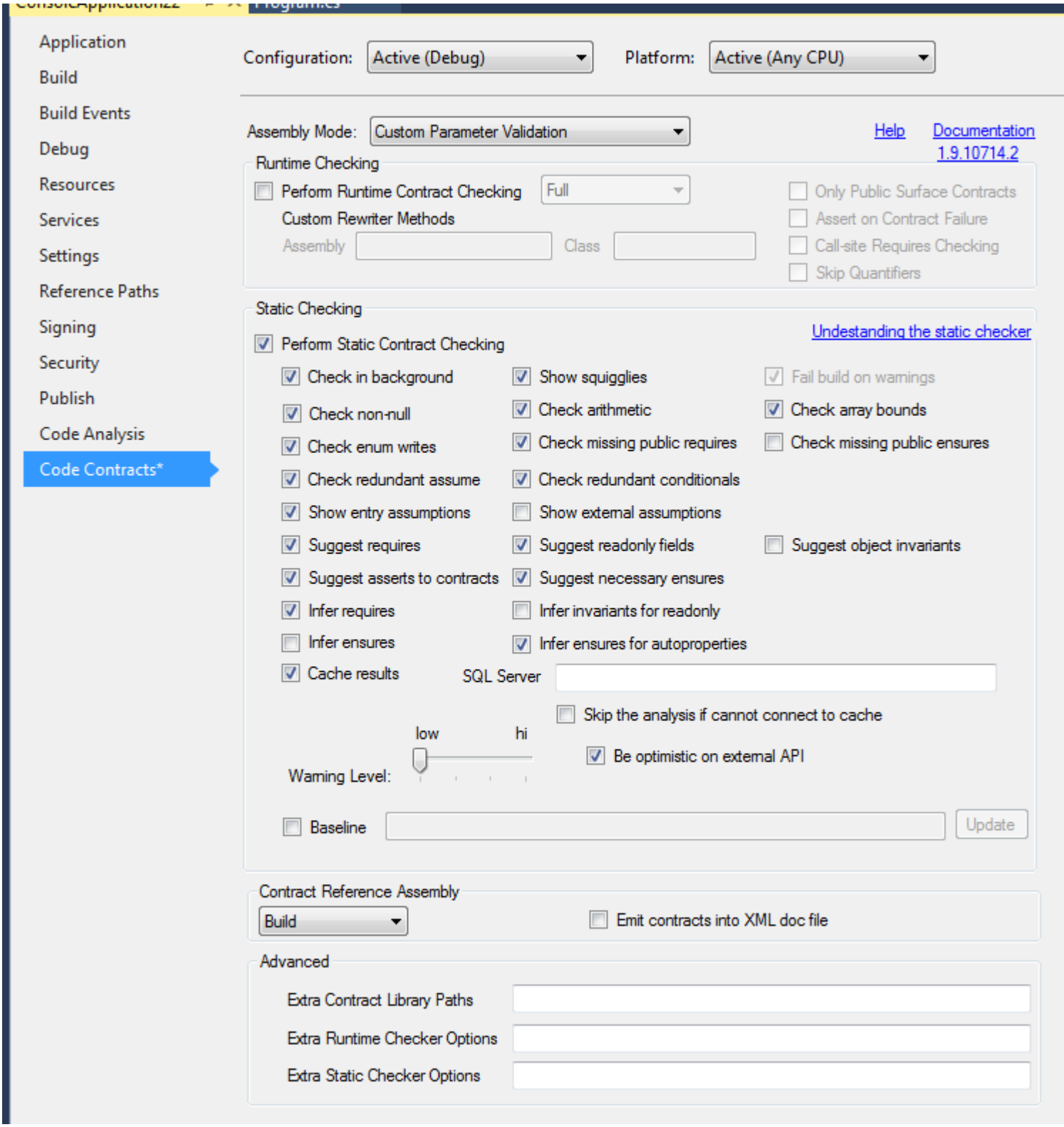
Installazione e attivazione dei contratti di codice

Mentre `System.Diagnostics.Contracts` è incluso all'interno di .Net Framework. Per utilizzare i Contratti di codice è necessario installare le estensioni di Visual Studio.

Sotto `Extensions and Updates` cerca i `Code Contracts` quindi installa gli `Code Contracts Tools`



Dopo aver installato gli strumenti, è necessario abilitare i `Code Contracts` all'interno della soluzione di progetto. Al minimo probabilmente vuoi abilitare il `Static Checking` (controlla dopo la costruzione). Se stai implementando una libreria che verrà utilizzata da altre soluzioni, potresti prendere in considerazione anche l'abilitazione del `Runtime Checking` .



Leggi Contratti di codice online: https://riptutorial.com/it/dot-net/topic/1937/contratti-di-codice

Capitolo 12: Crittografia / Crittografia

Osservazioni

.NET Framework fornisce l'implementazione di molti algoritmi crittografici. Includono algoritmi simmetrici, algoritmi asimmetrici e hash.

Examples

RijndaelManaged

Namespace richiesto: `System.Security.Cryptography`

```
private class Encryption {

          private const string SecretKey = "topSecretKeyusedforEncryptions";

          private const string SecretIv = "secretVectorHere";

          public string Encrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
Convert.ToBase64String(this.EncryptStringToBytesAes(data, this.GetCryptographyKey(),
this.GetCryptographyIv()));
          }

          public string Decrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
this.DecryptStringFromBytesAes(Convert.FromBase64String(data), this.GetCryptographyKey(),
this.GetCryptographyIv());
          }

          private byte[] GetCryptographyKey() {
            return Encoding.ASCII.GetBytes(SecretKey.Replace('e', '!'));
          }

          private byte[] GetCryptographyIv() {
            return Encoding.ASCII.GetBytes(SecretIv.Replace('r', '!'));
          }

          private byte[] EncryptStringToBytesAes(string plainText, byte[] key, byte[] iv) {
            MemoryStream encrypt;
            RijndaelManaged aesAlg = null;
            try {
              aesAlg = new RijndaelManaged {
                Key = key,
                IV = iv
              };
              var encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
              encrypt = new MemoryStream();
              using (var csEncrypt = new CryptoStream(encrypt, encryptor,
CryptoStreamMode.Write)) {
                using (var swEncrypt = new StreamWriter(csEncrypt)) {
                  swEncrypt.Write(plainText);
                }
```

```
            }
          } finally {
            aesAlg?.Clear();
          }
          return encrypt.ToArray();
        }

        private string DecryptStringFromBytesAes(byte[] cipherText, byte[] key, byte[] iv) {
          RijndaelManaged aesAlg = null;
          string plaintext;
          try {
            aesAlg = new RijndaelManaged {
              Key = key,
              IV = iv
            };
            var decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
            using (var msDecrypt = new MemoryStream(cipherText)) {
              using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read)) {
                using (var srDecrypt = new StreamReader(csDecrypt))
                  plaintext = srDecrypt.ReadToEnd();
              }
            }
          } finally {
            aesAlg?.Clear();
          }
          return plaintext;
        }
      }
```

uso

```
var textToEncrypt = "hello World";

 var encrypted = new Encryption().Encrypt(textToEncrypt); //-> zBmW+FUxOvdbpOGm9Ss/vQ==

 var decrypted = new Encryption().Decrypt(encrypted); //-> hello World
```

Nota:

- Rijndael è il predecessore dell'algoritmo crittografico simmetrico standard AES.

Criptare e decrittografare i dati utilizzando AES (in C #)

```
using System;
using System.IO;
using System.Security.Cryptography;

namespace Aes_Example
{
    class AesExample
    {
        public static void Main()
        {
            try
            {
                string original = "Here is some data to encrypt!";
```

```
            // Create a new instance of the Aes class.
            // This generates a new key and initialization vector (IV).
            using (Aes myAes = Aes.Create())
            {
                // Encrypt the string to an array of bytes.
                byte[] encrypted = EncryptStringToBytes_Aes(original,
                                                            myAes.Key,
                                                            myAes.IV);

                // Decrypt the bytes to a string.
                string roundtrip = DecryptStringFromBytes_Aes(encrypted,
                                                              myAes.Key,
                                                              myAes.IV);

                //Display the original data and the decrypted data.
                Console.WriteLine("Original:   {0}", original);
                Console.WriteLine("Round Trip: {0}", roundtrip);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: {0}", e.Message);
        }
    }

    static byte[] EncryptStringToBytes_Aes(string plainText, byte[] Key, byte[] IV)
    {
        // Check arguments.
        if (plainText == null || plainText.Length <= 0)
            throw new ArgumentNullException("plainText");
        if (Key == null || Key.Length <= 0)
            throw new ArgumentNullException("Key");
        if (IV == null || IV.Length <= 0)
            throw new ArgumentNullException("IV");

        byte[] encrypted;

        // Create an Aes object with the specified key and IV.
        using (Aes aesAlg = Aes.Create())
        {
            aesAlg.Key = Key;
            aesAlg.IV = IV;

            // Create a decrytor to perform the stream transform.
            ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key,
                                                                aesAlg.IV);

            // Create the streams used for encryption.
            using (MemoryStream msEncrypt = new MemoryStream())
            {
                using (CryptoStream csEncrypt = new CryptoStream(msEncrypt,
                                                                 encryptor,
                                                                 CryptoStreamMode.Write))
                {
                    using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
                    {
                        //Write all data to the stream.
                        swEncrypt.Write(plainText);
                    }
```

```
                        encrypted = msEncrypt.ToArray();
                    }
                }
            }

            // Return the encrypted bytes from the memory stream.
            return encrypted;
        }

        static string DecryptStringFromBytes_Aes(byte[] cipherText, byte[] Key, byte[] IV)
        {
            // Check arguments.
            if (cipherText == null || cipherText.Length <= 0)
                throw new ArgumentNullException("cipherText");
            if (Key == null || Key.Length <= 0)
                throw new ArgumentNullException("Key");
            if (IV == null || IV.Length <= 0)
                throw new ArgumentNullException("IV");

            // Declare the string used to hold the decrypted text.
            string plaintext = null;

            // Create an Aes object with the specified key and IV.
            using (Aes aesAlg = Aes.Create())
            {
                aesAlg.Key = Key;
                aesAlg.IV = IV;

                // Create a decrytor to perform the stream transform.
                ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key,
                                                                    aesAlg.IV);

                // Create the streams used for decryption.
                using (MemoryStream msDecrypt = new MemoryStream(cipherText))
                {
                    using (CryptoStream csDecrypt = new CryptoStream(msDecrypt,
                                                                     decryptor,
                                                                     CryptoStreamMode.Read))
                    {
                        using (StreamReader srDecrypt = new StreamReader(csDecrypt))
                        {

                            // Read the decrypted bytes from the decrypting stream
                            // and place them in a string.
                            plaintext = srDecrypt.ReadToEnd();
                        }
                    }
                }
            }

            return plaintext;
        }
    }
}
```

Questo esempio è da MSDN .

È un'applicazione demo della console che mostra come crittografare una stringa utilizzando la crittografia **AES** standard e come decrittografarla in seguito.

(**AES = Advanced Encryption Standard** , una specifica per la crittografia dei dati elettronici stabilita dal National Institute of Standards and Technology (NIST) nel 2001, che è ancora lo standard de facto per la crittografia simmetrica)

Gli appunti:

- In uno scenario di crittografia reale, è necessario scegliere una modalità di crittografia appropriata (può essere assegnata alla proprietà `Mode` selezionando un valore `CipherMode`). **Non** utilizzare **mai** `CipherMode.ECB` (modalità del libro di codici elettronico), poiché ciò genera un flusso debole di cypher
- Per creare una buona (e non un debole) `Key` , utilizzare un generatore casuale di crittografia o utilizzare l'esempio precedente **(Creare una chiave da una password).** Il **KeySize** consigliato è 256 bit. Le dimensioni chiave supportate sono disponibili tramite la proprietà `LegalKeySizes` .
- Per inizializzare il vettore di inizializzazione `IV` , puoi usare un SALT come mostrato nell'esempio sopra (**Random SALT**)
- Le dimensioni del blocco supportate sono disponibili tramite la proprietà `SupportedBlockSizes` , la dimensione del blocco può essere assegnata tramite la proprietà `BlockSize`

Uso: vedi il metodo Main ().

Crea una chiave da una password / SALE casuale (in C #)

```
using System;
using System.Security.Cryptography;
using System.Text;

public class PasswordDerivedBytesExample
{
    public static void Main(String[] args)
    {
        // Get a password from the user.
        Console.WriteLine("Enter a password to produce a key:");

        byte[] pwd = Encoding.Unicode.GetBytes(Console.ReadLine());

        byte[] salt = CreateRandomSalt(7);

        // Create a TripleDESCryptoServiceProvider object.
        TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();

        try
        {
            Console.WriteLine("Creating a key with PasswordDeriveBytes...");

            // Create a PasswordDeriveBytes object and then create
            // a TripleDES key from the password and salt.
            PasswordDeriveBytes pdb = new PasswordDeriveBytes(pwd, salt);

            // Create the key and set it to the Key property
            // of the TripleDESCryptoServiceProvider object.
```

```
            tdes.Key = pdb.CryptDeriveKey("TripleDES", "SHA1", 192, tdes.IV);

            Console.WriteLine("Operation complete.");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            // Clear the buffers
            ClearBytes(pwd);
            ClearBytes(salt);

            // Clear the key.
            tdes.Clear();
        }

        Console.ReadLine();
    }

    #region Helper methods

    /// <summary>
    /// Generates a random salt value of the specified length.
    /// </summary>
    public static byte[] CreateRandomSalt(int length)
    {
        // Create a buffer
        byte[] randBytes;

        if (length >= 1)
        {
            randBytes = new byte[length];
        }
        else
        {
            randBytes = new byte[1];
        }

        // Create a new RNGCryptoServiceProvider.
        RNGCryptoServiceProvider rand = new RNGCryptoServiceProvider();

        // Fill the buffer with random bytes.
        rand.GetBytes(randBytes);

        // return the bytes.
        return randBytes;
    }

    /// <summary>
    /// Clear the bytes in a buffer so they can't later be read from memory.
    /// </summary>
    public static void ClearBytes(byte[] buffer)
    {
        // Check arguments.
        if (buffer == null)
        {
            throw new ArgumentNullException("buffer");
        }
```

```
        // Set each byte in the buffer to 0.
        for (int x = 0; x < buffer.Length; x++)
        {
            buffer[x] = 0;
        }
    }

    #endregion
}
```

Questo esempio è preso da MSDN.

È una demo della console e mostra come creare una chiave sicura basata su una password definita dall'utente e come creare un SALE casuale basato sul generatore casuale crittografico.

Gli appunti:

- La funzione incorporata `PasswordDeriveBytes` utilizza l'algoritmo PBKDF1 standard per generare una chiave dalla password. Per impostazione predefinita, utilizza 100 iterazioni per generare la chiave per rallentare gli attacchi di forza bruta. Il SALT generato a caso rinforza ulteriormente la chiave.
- La funzione `CryptDeriveKey` converte la chiave generata da `PasswordDeriveBytes` in una chiave compatibile con l'algoritmo di crittografia specificato (qui "TripleDES") utilizzando l'algoritmo hash specificato (qui "SHA1"). La chiave in questo esempio è 192 byte e il vettore di inizializzazione IV viene preso dal provider di crittografia triple-DES
- Di solito, questo meccanismo viene utilizzato per proteggere una chiave generata a caso più forte da una password, che crittografa una grande quantità di dati. È inoltre possibile utilizzarlo per fornire più password di utenti diversi per consentire l'accesso agli stessi dati (protetti da una chiave casuale diversa).
- Sfortunatamente, `CryptDeriveKey` attualmente non supporta AES. Vedi qui
  NOTA: Come soluzione alternativa, è possibile creare una chiave AES casuale per la crittografia dei dati da proteggere con AES e memorizzare la chiave AES in un contenitore TripleDES che utilizza la chiave generata da `CryptDeriveKey` . Ma ciò limita la sicurezza a TripleDES, non sfrutta le chiavi più grandi di AES e crea una dipendenza da TripleDES.

Utilizzo: vedi il metodo Main ().

Crittografia e decrittografia tramite Crittografia (AES)

Codice di decrittografia

```
    public static string Decrypt(string cipherText)
    {
        if (cipherText == null)
            return null;

        byte[] cipherBytes = Convert.FromBase64String(cipherText);
        using (Aes encryptor = Aes.Create())
        {
```

```
            Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
            encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateDecryptor(),
CryptoStreamMode.Write))
                {
                    cs.Write(cipherBytes, 0, cipherBytes.Length);
                    cs.Close();
                }

                cipherText = Encoding.Unicode.GetString(ms.ToArray());
            }
        }

        return cipherText;
    }
```

Codice di crittografia

```
    public static string Encrypt(string cipherText)
    {
        if (cipherText == null)
            return null;

        byte[] clearBytes = Encoding.Unicode.GetBytes(cipherText);
        using (Aes encryptor = Aes.Create())
        {
            Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
            encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateEncryptor(),
CryptoStreamMode.Write))
                {
                    cs.Write(clearBytes, 0, clearBytes.Length);
                    cs.Close();
                }

                cipherText = Convert.ToBase64String(ms.ToArray());
            }
        }
        return cipherText;
    }
```

uso

```
var textToEncrypt = "TestEncrypt";

var encrypted = Encrypt(textToEncrypt);

var decrypted = Decrypt(encrypted);
```

Leggi Crittografia / Crittografia online: https://riptutorial.com/it/dot-net/topic/7615/crittografia---crittografia

Capitolo 13: Data e ora di analisi

Examples

ParseExact

```
var dateString = "2015-11-24";

var date = DateTime.ParseExact(dateString, "yyyy-MM-dd", null);
Console.WriteLine(date);
```

24/11/2015 12:00:00

Si noti che il passaggio di `CultureInfo.CurrentCulture` come terzo parametro è identico al passaggio di `null` . Oppure, puoi passare una cultura specifica.

Formato stringhe

La stringa di input può essere in qualsiasi formato che corrisponda alla stringa di formato

```
var date = DateTime.ParseExact("24|201511", "dd|yyyyMM", null);
Console.WriteLine(date);
```

24/11/2015 12:00:00

Tutti i caratteri che non sono specificatori di formato sono trattati come valori letterali

```
var date = DateTime.ParseExact("2015|11|24", "yyyy|MM|dd", null);
Console.WriteLine(date);
```

24/11/2015 12:00:00

Il caso vale per gli specificatori di formato

```
var date = DateTime.ParseExact("2015-01-24 11:11:30", "yyyy-mm-dd hh:MM:ss", null);
Console.WriteLine(date);
```

11/24/2015 11:01:30 AM

Si noti che i valori del mese e dei minuti sono stati analizzati nelle destinazioni sbagliate.

Le stringhe di formato a carattere singolo devono essere uno dei formati standard

```
var date = DateTime.ParseExact("11/24/2015", "d", new CultureInfo("en-US"));
var date = DateTime.ParseExact("2015-11-24T10:15:45", "s", null);
var date = DateTime.ParseExact("2015-11-24 10:15:45Z", "u", null);
```

eccezioni

ArgumentNullException

```
var date = DateTime.ParseExact(null, "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", null, null);
```

FormatException

```
var date = DateTime.ParseExact("", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "", null);
var date = DateTime.ParseExact("2015-0C-24", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "yyyy-QQ-dd", null);

// Single-character format strings must be one of the standard formats
var date = DateTime.ParseExact("2015-11-24", "q", null);

// Format strings must match the input exactly* (see next section)
var date = DateTime.ParseExact("2015-11-24", "d", null); // Expects 11/24/2015 or 24/11/2015
for most cultures
```

Gestire più formati possibili

```
var date = DateTime.ParseExact("2015-11-24T10:15:45",
  new [] { "s", "t", "u", "yyyy-MM-dd" }, // Will succeed as long as input matches one of
these
  CultureInfo.CurrentCulture, DateTimeStyles.None);
```

Gestire le differenze culturali

```
var dateString = "10/11/2015";
var date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-US"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

Giorno: 11; Mese: 10

```
date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-GB"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

Giorno: 10; Mese: 11

TryParse

Questo metodo accetta una stringa come input, tenta di analizzarla in un `DateTime` e restituisce un risultato booleano che indica il successo o l'insuccesso. Se la chiamata ha esito positivo, la variabile viene trasmessa quando il parametro `out` viene popolato con il risultato analizzato.

Se l'analisi fallisce, la variabile passa come il parametro `out` è impostato sul valore predefinito, `DateTime.MinValue` .

TryParse (string, out DateTime)

```
DateTime parsedValue;
```

```
if (DateTime.TryParse("monkey", out parsedValue))
{
   Console.WriteLine("Apparently, 'monkey' is a date/time value. Who knew?");
}
```

Questo metodo tenta di analizzare la stringa di input in base alle impostazioni internazionali del sistema e ai formati noti come ISO 8601 e altri formati comuni.

```
DateTime.TryParse("11/24/2015 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24T14:28:42", out parsedValue); // true
DateTime.TryParse("Sat, 24 Nov 2015 14:28:42", out parsedValue); // true
```

Poiché questo metodo non accetta informazioni sulla cultura, utilizza le impostazioni locali del sistema. Questo può portare a risultati inaspettati.

```
// System set to en-US culture
bool result = DateTime.TryParse("24/11/2015", out parsedValue);
Console.WriteLine(result);
```

falso

```
// System set to en-GB culture
bool result = DateTime.TryParse("11/24/2015", out parsedValue);
Console.WriteLine(result);
```

falso

```
// System set to en-GB culture
bool result = DateTime.TryParse("10/11/2015", out parsedValue);
Console.WriteLine(result);
```

Vero

Nota che se sei negli Stati Uniti, potresti essere sorpreso che il risultato analizzato sia il 10 novembre, non l'11 ottobre.

TryParse (string, IFormatProvider, DateTimeStyles, out DateTime)

```
if (DateTime.TryParse(" monkey ", new CultureInfo("en-GB"),
    DateTimeStyles.AllowLeadingWhite | DateTimeStyles.AllowTrailingWhite, out parsedValue)
{
    Console.WriteLine("Apparently, ' monkey ' is a date/time value. Who knew?");
}
```

A differenza del metodo fratello, questo overload consente di specificare una cultura e uno o più stili specifici. Il passaggio `null` per il parametro `IFormatProvider` utilizza la cultura di sistema.

eccezioni

Si noti che è possibile che questo metodo generi un'eccezione in determinate condizioni. Questi si riferiscono ai parametri introdotti per questo overload: `IFormatProvider` e `DateTimeStyles` .

- `NotSupportedException` : `IFormatProvider` specifica una lingua neutra
- `ArgumentException` : `DateTimeStyles` non è un'opzione valida o contiene flag incompatibili come `AssumeLocal` e `AssumeUniversal` .

TryParseExact

Questo metodo si comporta come una combinazione di `TryParse` e `ParseExact` : consente di specificare il / i formato / i personalizzato e restituisce un risultato booleano che indica il successo o l'insuccesso anziché lanciare un'eccezione se l'analisi fallisce.

TryParseExact (stringa, stringa, IFormatProvider, DateTimeStyles, out DateTime)

Questo overload tenta di analizzare la stringa di input su un formato specifico. La stringa di input deve corrispondere a quel formato per poter essere analizzata.

```
DateTime.TryParseExact("11242015", "MMddyyyy", null, DateTimeStyles.None, out parsedValue); //
true
```

TryParseExact (string, string [], IFormatProvider, DateTimeStyles, out DateTime)

Questo overload tenta di analizzare la stringa di input su una serie di formati. La stringa di input deve corrispondere ad almeno un formato per poter essere analizzata.

```
DateTime.TryParseExact("11242015", new [] { "yyyy-MM-dd", "MMddyyyy" }, null,
DateTimeStyles.None, out parsedValue); // true
```

Leggi Data e ora di analisi online: https://riptutorial.com/it/dot-net/topic/58/data-e-ora-di-analisi

Capitolo 14: dizionari

Examples

Enumerazione di un dizionario

Puoi enumerare attraverso un dizionario in uno dei 3 modi:

Utilizzo delle coppie KeyValue

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(KeyValuePair<int, string> kvp in dict)
{
   Console.WriteLine("Key : " + kvp.Key.ToString() + ", Value : " + kvp.Value);
}
```

Usando le chiavi

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(int key in dict.Keys)
{
    Console.WriteLine("Key : " + key.ToString() + ", Value : " + dict[key]);
}
```

Utilizzo dei valori

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(string s in dict.Values)
{
    Console.WriteLine("Value : " + s);
}
```

Inizializzazione di un dizionario con un inizializzatore di raccolta

```
// Translates to `dict.Add(1, "First")` etc.
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};

// Translates to `dict[1] = "First"` etc.
// Works in C# 6.0.
var dict = new Dictionary<int, string>()
{
    [1] = "First",
    [2] = "Second",
    [3] = "Third"
};
```

Aggiunta a un dizionario

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict.Add(1, "First");
dict.Add(2, "Second");

// To safely add items (check to ensure item does not already exist - would throw)
if(!dict.ContainsKey(3))
{
   dict.Add(3, "Third");
}
```

In alternativa, possono essere aggiunti / impostati tramite l'indicizzatore. (Un indicizzatore internamente ha l'aspetto di una proprietà, con un get e un set, ma accetta un parametro di qualsiasi tipo specificato tra parentesi):

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict[1] = "First";
dict[2] = "Second";
dict[3] = "Third";
```

A differenza del metodo `Add` che genera un'eccezione, se una chiave è già contenuta nel dizionario, l'indicizzatore sostituisce semplicemente il valore esistente.

Per il dizionario thread-safe utilizzare `ConcurrentDictionary<TKey, TValue>` :

```
var dict = new ConcurrentDictionary<int, string>();
dict.AddOrUpdate(1, "First", (oldKey, oldValue) => "First");
```

Ottenere un valore da un dizionario

Dato questo codice di installazione:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```

Potresti voler leggere il valore per la voce con la chiave 1. Se la chiave non esiste, ottenere un valore genererà `KeyNotFoundException` , quindi potresti voler prima verificarlo con `ContainsKey` :

```
if (dict.ContainsKey(1))
    Console.WriteLine(dict[1]);
```

Questo ha uno svantaggio: cercherete nel vostro dizionario due volte (una volta per verificare l'esistenza e una per leggere il valore). Per un dizionario di grandi dimensioni questo può influire sulle prestazioni. Fortunatamente entrambe le operazioni possono essere eseguite insieme:

```
string value;
if (dict.TryGetValue(1, out value))
    Console.WriteLine(value);
```

Crea un dizionario con chiavi Case-Insensivitve.

```
var MyDict = new Dictionary<string,T>(StringComparison.InvariantCultureIgnoreCase)
```

ConcurrentDictionary (da .NET 4.0)

> Rappresenta una raccolta thread-safe di coppie chiave / valore a cui è possibile accedere simultaneamente da più thread.

Creare un'istanza

La creazione di un'istanza funziona più o meno allo stesso modo con il `Dictionary<TKey, TValue>` , ad esempio:

```
var dict = new ConcurrentDictionary<int, string>();
```

Aggiunta o aggiornamento

Potresti essere sorpreso dal fatto che non esiste un metodo `Add` , ma invece c'è `AddOrUpdate` con 2 overload:

(1) `AddOrUpdate(TKey key, TValue, Func<TKey, TValue, TValue> addValue)` - *Aggiunge una coppia chiave / valore se la chiave non esiste già o aggiorna una coppia chiave / valore utilizzando la funzione specificata se la chiave esiste già.*

(2) `AddOrUpdate(TKey key, Func<TKey, TValue> addValue, Func<TKey, TValue, TValue> updateValueFactory)` - *Usa le funzioni specificate per aggiungere una coppia chiave / valore al se la chiave non esiste già, o per aggiornare una coppia chiave / valore se la chiave esiste già.*

Aggiunta o aggiornamento di un valore, indipendentemente dal valore se era già presente per la chiave data (1):

```
string addedValue = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => "First");
```

Aggiunta o aggiornamento di un valore, ma ora modifica il valore in aggiornamento, in base al valore precedente (1):

```
string addedValue2 = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => $"{valueOld}
Updated");
```

Usando il sovraccarico (2) possiamo anche aggiungere un nuovo valore usando una fabbrica:

```
string addedValue3 = dict.AddOrUpdate(1, (key) => key == 1 ? "First" : "Not First",
(updateKey, valueOld) => $"{valueOld} Updated");
```

Ottenere valore

Ottenere un valore è lo stesso del `Dictionary<TKey,TValue>` :

```
string value = null;
bool success = dict.TryGetValue(1, out value);
```

Ottenere o aggiungere un valore

Ci sono due overload di mehod, che **otterranno o aggiungeranno** un valore in modo thread-safe.

Ottieni valore con la chiave 2 o aggiungi il valore "Second" se la chiave non è presente:

```
string theValue = dict.GetOrAdd(2, "Second");
```

Utilizzo di una factory per l'aggiunta di un valore, se il valore non è presente:

```
string theValue2 = dict.GetOrAdd(2, (key) => key == 2 ? "Second" : "Not Second." );
```

IEnumerable to Dictionary (≥ .NET 3.5)

Crea un dizionario <TKey, TValue> da un oggetto IEnumerable <T> :

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

```
public class Fruits
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

```
var fruits = new[]
{
    new Fruits { Id = 8 , Name = "Apple" },
    new Fruits { Id = 3 , Name = "Banana" },
    new Fruits { Id = 7 , Name = "Mango" },
};


// Dictionary<int, string>                  key      value
var dictionary = fruits.ToDictionary(x => x.Id, x => x.Name);
```

Rimozione da un dizionario

Dato questo codice di installazione:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```

Utilizzare il metodo `Remove` per rimuovere una chiave e il relativo valore associato.

```
bool wasRemoved = dict.Remove(2);
```

L'esecuzione di questo codice rimuove la chiave `2` e il suo valore dal dizionario. `Remove` restituisce un valore booleano che indica se la chiave specificata è stata trovata e rimossa dal dizionario. Se la chiave non esiste nel dizionario, nulla viene rimosso dal dizionario e viene restituito false (non viene generata alcuna eccezione).

Non è **corretto** provare e rimuovere una chiave impostando il valore per la chiave su `null` .

```
dict[2] = null; // WRONG WAY TO REMOVE!
```

Questo non rimuoverà la chiave. Sostituirà semplicemente il valore precedente con un valore `null` .

Per rimuovere tutte le chiavi e i valori da un dizionario, utilizzare il metodo `Clear` .

```
dict.Clear();
```

Dopo l'esecuzione di `Clear` , il `Count` del dizionario sarà 0, ma la capacità interna rimane invariata.

ContainsKey (TKey)

Per verificare se un `Dictionary` ha una chiave specifica, puoi chiamare il metodo `ContainsKey(TKey)` e fornire la chiave del tipo `TKey` . Il metodo restituisce un valore `bool` quando la chiave esiste sul dizionario. Per esempio:

```
var dictionary = new Dictionary<string, Customer>()
{
   {"F1", new Customer() { FirstName = "Felipe", ... } },
   {"C2", new Customer() { FirstName = "Carl", ... } },
   {"J7", new Customer() { FirstName = "John", ... } },
   {"M5", new Customer() { FirstName = "Mary", ... } },
};
```

E controlla se esiste un `C2` sul dizionario:

```
if (dictionary.ContainsKey("C2"))
{
   // exists
```

```
}
```

Il metodo ContainsKey è disponibile nella versione generica `Dictionary<TKey, TValue>` .

Dizionario alla lista

Creazione di un elenco di KeyValuePair:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<KeyValuePair<int, int>> list = new List<KeyValuePair<int, int>>();
list.AddRange(dictionary);
```

Creazione di un elenco di chiavi:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Keys);
```

Creazione di un elenco di valori:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Values);
```

ConcurrentDictionary aumentato con Lazy'1 riduce il calcolo duplicato

Problema

ConcurrentDictionary brilla quando si tratta di restituire istantaneamente chiavi esistenti dalla cache, per lo più senza blocco e contendenti a livello granulare. Ma cosa succede se la creazione dell'oggetto è davvero costosa, superando il costo del cambio di contesto e si verificano alcuni errori di cache?

Se viene richiesta la stessa chiave da più thread, alla fine verrà aggiunto uno degli oggetti risultanti dalle operazioni di collisione e gli altri verranno gettati via, sprecando la risorsa CPU per creare l'oggetto e la risorsa di memoria per archiviare temporaneamente l'oggetto . Altre risorse potrebbero essere sprecate pure. Questo è veramente brutto.

Soluzione

Possiamo combinare `ConcurrentDictionary<TKey, TValue>` con `Lazy<TValue>` . L'idea è che il metodo GetOrAdd ConcurrentDictionary può solo restituire il valore che è stato effettivamente aggiunto alla collezione. Anche gli oggetti Lazy che perdono potrebbero essere sprecati in questo caso, ma non è un problema, dato che l'oggetto Lazy è relativamente poco costoso. La proprietà Value del Lazy perdente non viene mai richiesta, perché siamo intelligenti nel richiedere solo la proprietà Value di quella effettivamente aggiunta alla raccolta, quella restituita dal metodo GetOrAdd:

```
public static class ConcurrentDictionaryExtensions
{
    public static TValue GetOrCreateLazy<TKey, TValue>(
        this ConcurrentDictionary<TKey, Lazy<TValue>> d,
        TKey key,
        Func<TKey, TValue> factory)
    {
        return
            d.GetOrAdd(
                key,
                key1 =>
                    new Lazy<TValue>(() => factory(key1),
                    LazyThreadSafetyMode.ExecutionAndPublication)).Value;
    }
}
```

La memorizzazione nella cache degli oggetti XmlSerializer può essere particolarmente costosa e anche all'avvio dell'applicazione vi sono molti conflitti. E c'è di più: se si tratta di serializzatori personalizzati, ci sarà una perdita di memoria anche per il resto del ciclo di vita del processo. L'unico vantaggio di ConcurrentDictionary in questo caso è che per il resto del ciclo di vita del processo non ci saranno blocchi, ma l'avvio dell'applicazione e l'utilizzo della memoria sarebbero inaccettabili. Questo è un lavoro per il nostro ConcurrentDictionary, aumentato con Lazy:

```
private ConcurrentDictionary<Type, Lazy<XmlSerializer>> _serializers =
    new ConcurrentDictionary<Type, Lazy<XmlSerializer>>();

public XmlSerializer GetSerialier(Type t)
{
    return _serializers.GetOrCreateLazy(t, BuildSerializer);
}

private XmlSerializer BuildSerializer(Type t)
{
    throw new NotImplementedException("and this is a homework");
}
```

Leggi dizionari online: https://riptutorial.com/it/dot-net/topic/45/dizionari

Capitolo 15: eccezioni

Osservazioni

Relazionato:

- MSDN: gestione delle eccezioni e delle eccezioni (C # Programming Guide)
- MSDN: gestione e eccezioni di lancio
- MSDN: CA1031: non rilevare i tipi di eccezione generici
- MSDN: try-catch (riferimento C #)

Examples

Cattura un'eccezione

Il codice può e deve generare eccezioni in circostanze eccezionali. Esempi di questo includono:

- Tentativo di leggere oltre la fine di un flusso
- Non disponendo delle autorizzazioni necessarie per accedere a un file
- Tentativo di eseguire un'operazione non valida, ad esempio la divisione per zero
- Un timeout che si verifica quando si scarica un file da Internet

Il chiamante può gestire queste eccezioni "catturandole" e dovrebbe farlo solo quando:

- Può effettivamente risolvere la circostanza eccezionale o recuperare in modo appropriato, oppure;
- Può fornire un contesto aggiuntivo all'eccezione che sarebbe utile se l'eccezione deve essere ripetuta (le eccezioni ridistribuite vengono catturate dai gestori di eccezioni più in alto nello stack di chiamate)

Va notato che la scelta di *non* prendere un'eccezione è perfettamente valida se l'intenzione è quella di essere gestita ad un livello più alto.

La cattura di un'eccezione viene eseguita avvolgendo il codice potenzialmente lancio in un blocco `try { ... }` come segue e rilevando le eccezioni che è in grado di gestire in un `catch (ExceptionType) { ... }`:

```
Console.Write("Please enter a filename: ");
string filename = Console.ReadLine();

Stream fileStream;

try
{
    fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
```

```
    Console.WriteLine("File '{0}' could not be found.", filename);
}
```

Usando un blocco finalmente

Il blocco `finally { ... }` di `try-finally` o `try-catch-finally` verrà sempre eseguito, indipendentemente dal fatto che si sia verificata o meno un'eccezione (tranne quando è stata generata una `StackOverflowException` o è stata effettuata una chiamata a `Environment.FailFast()`).

Può essere utilizzato per liberare o ripulire le risorse acquisite nel blocco `try { ... }` modo sicuro.

```
Console.Write("Please enter a filename: ");
string filename = Console.ReadLine();

Stream fileStream = null;

try
{
    fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
    Console.WriteLine("File '{0}' could not be found.", filename);
}
finally
{
    if (fileStream != null)
    {
        fileStream.Dispose();
    }
}
```

Cattura e rilancio hanno catturato eccezioni

Quando si desidera rilevare un'eccezione e fare qualcosa, ma non è possibile continuare l'esecuzione del blocco di codice corrente a causa dell'eccezione, è possibile che si desideri rilanciare l'eccezione al successivo gestore di eccezioni nello stack di chiamate. Ci sono buoni modi e cattivi modi per farlo.

```
private static void AskTheUltimateQuestion()
{
    try
    {
        var x = 42;
        var y = x / (x - x); // will throw a DivideByZeroException

        // IMPORTANT NOTE: the error in following string format IS intentional
        // and exists to throw an exception to the FormatException catch, below
        Console.WriteLine("The secret to life, the universe, and everything is {1}", y);
    }
    catch (DivideByZeroException)
    {
        // we do not need a reference to the exception
        Console.WriteLine("Dividing by zero would destroy the universe.");
```

```
        // do this to preserve the stack trace:
        throw;
    }
    catch (FormatException ex)
    {
        // only do this if you need to change the type of the Exception to be thrown
        // and wrap the inner Exception

        // remember that the stack trace of the outer Exception will point to the
        // next line

        // you'll need to examine the InnerException property to get the stack trace
        // to the line that actually started the problem

        throw new InvalidOperationException("Watch your format string indexes.", ex);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Something else horrible happened. The exception: " + ex.Message);

        // do not do this, because the stack trace will be changed to point to
        // this location instead of the location where the exception
        // was originally thrown:
        throw ex;
    }
}

static void Main()
{
    try
    {
        AskTheUltimateQuestion();
    }
    catch
    {
        // choose this kind of catch if you don't need any information about
        // the exception that was caught

        // this block "eats" all exceptions instead of rethrowing them
    }
}
```

È possibile filtrare per tipo di eccezione e anche per proprietà di eccezione (nuovo in C # 6.0, un po 'più lungo disponibile in VB.NET (citazione necessaria)):

Documentazione / C # / nuove funzionalità

Filtri di eccezione

Poiché le eccezioni C # 6.0 possono essere filtrate usando l'operatore `when` .

Questo è simile all'utilizzo di un semplice `if` non si srotola lo stack se la condizione all'interno di `when` non è soddisfatta.

Esempio

```
try
```

```
{
  // ...
}
catch (Exception e) when (e.InnerException != null) // Any condition can go in here.
{
  // ...
}
```

Le stesse informazioni si possono trovare nelle Funzionalità C # 6.0 qui: Filtri di eccezione

Ripartire un'eccezione all'interno di un blocco catch

All'interno di un blocco `catch` , la parola chiave `throw` può essere utilizzata da sola, senza specificare un valore di eccezione, per *ripensare* all'eccezione appena rilevata. Il richiamo di un'eccezione consente all'eccezione originale di continuare la catena di gestione delle eccezioni, preservando lo stack di chiamata oi dati associati:

```
try {...}
catch (Exception ex) {
  // Note: the ex variable is *not* used
  throw;
}
```

Un anti-pattern comune è invece quello di `throw ex` , che ha l'effetto di limitare la vista del prossimo gestore di eccezioni della traccia dello stack:

```
try {...}
catch (Exception ex) {
  // Note: the ex variable is thrown
  //  future stack traces of the exception will not see prior calls
  throw ex;
}
```

In generale, utilizzando `throw ex` non è auspicabile, come i futuri gestori di eccezioni, che ispezionano l'analisi dello stack saranno in grado di vedere le chiamate fin dal `throw ex` . Omettendo la variabile `ex` e usando solo la parola chiave `throw` l'eccezione originale sarà "bubble-up" .

Lanciare un'eccezione da un metodo diverso preservando le sue informazioni

Occasionalmente potresti catturare un'eccezione e lanciarla da un thread o metodo diverso preservando lo stack delle eccezioni originale. Questo può essere fatto con `ExceptionDispatchInfo` :

```
using System.Runtime.ExceptionServices;

void Main()
{
    ExceptionDispatchInfo capturedException = null;
    try
    {
        throw new Exception();
    }
```

```
    catch (Exception ex)
    {
        capturedException = ExceptionDispatchInfo.Capture(ex);
    }

    Foo(capturedException);
}

void Foo(ExceptionDispatchInfo exceptionDispatchInfo)
{
    // Do stuff

    if (capturedException != null)
    {
        // Exception stack trace will show it was thrown from Main() and not from Foo()
        exceptionDispatchInfo.Throw();
    }
}
```

Leggi eccezioni online: https://riptutorial.com/it/dot-net/topic/33/eccezioni

Capitolo 16: Elaborazione parallela mediante framework .Net

introduzione

Questo argomento riguarda la programmazione multi core usando Task Parallel Library con .NET framework. La libreria parallela delle attività consente di scrivere codice leggibile dall'uomo e di adattarsi al numero di core disponibili. Quindi puoi essere sicuro che il tuo software si aggiornerebbe automaticamente con l'ambiente di aggiornamento.

Examples

Estensioni parallele

Sono state introdotte estensioni parallele insieme alla libreria parallela Task per ottenere il parallelismo dei dati. Il parallelismo dei dati fa riferimento a scenari in cui la stessa operazione viene eseguita contemporaneamente (ovvero, in parallelo) su elementi di una raccolta o matrice di origine. .NET fornisce nuovi costrutti per ottenere il parallelismo dei dati utilizzando i costrutti Parallel.For e Parallel.Foreach.

```
//Sequential version

foreach (var item in sourcecollection){

Process(item);

}

// Parallel equivalent

Parallel.foreach(sourcecollection, item => Process(item));
```

Il suddetto costrutto Parallel.ForEach utilizza i nuclei multipli e quindi migliora le prestazioni nello stesso modo.

Leggi Elaborazione parallela mediante framework .Net online: https://riptutorial.com/it/dot-net/topic/8085/elaborazione-parallela-mediante-framework--net

Capitolo 17: Espressioni regolari (System.Text.RegularExpressions)

Examples

Controlla se il modello corrisponde all'input

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern);
}
```

Passando Opzioni

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase | RegexOptions.Singleline);
}
```

Partita e sostituzione semplici

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"W.rld";

    // Hello Stack Overflow!
    return Regex.Replace(input, pattern, "Stack Overflow");
}
```

Abbina in gruppi

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. (?<Subject>W.rld)!";

    Match match = Regex.Match(input, pattern);

    // World
    return match.Groups["Subject"].Value;
```

```
}
```

Rimuovi caratteri non alfanumerici dalla stringa

```
public string Remove()
{
    string input = "Hello./!";

    return Regex.Replace(input, "[^a-zA-Z0-9]", "");
}
```

Trova tutte le partite

utilizzando

```
using System.Text.RegularExpressions;
```

Codice

```
static void Main(string[] args)
{
    string input = "Carrot Banana Apple Cherry Clementine Grape";
    // Find words that start with uppercase 'C'
    string pattern = @"\bC\w*\b";

    MatchCollection matches = Regex.Matches(input, pattern);
    foreach (Match m in matches)
        Console.WriteLine(m.Value);
}
```

Produzione

```
Carrot
Cherry
Clementine
```

Leggi Espressioni regolari (System.Text.RegularExpressions) online: https://riptutorial.com/it/dot-net/topic/6944/espressioni-regolari--system-text-regularexpressions-

Capitolo 18: File Input / Output

Parametri

Parametro	Dettagli
percorso di stringa	Percorso del file da controllare. (relativo o pienamente qualificato)

Osservazioni

Restituisce true se il file esiste, false altrimenti.

Examples

VB WriteAllText

```
Imports System.IO

Dim filename As String = "c:\path\to\file.txt"
File.WriteAllText(filename, "Text to write" & vbCrLf)
```

VB StreamWriter

```
Dim filename As String = "c:\path\to\file.txt"
If System.IO.File.Exists(filename) Then
    Dim writer As New System.IO.StreamWriter(filename)
    writer.Write("Text to write" & vbCrLf) 'Add a newline
    writer.close()
End If
```

C # StreamWriter

```
using System.Text;
using System.IO;

string filename = "c:\path\to\file.txt";
//'using' structure allows for proper disposal of stream.
using (StreamWriter writer = new StreamWriter(filename"))
{
    writer.WriteLine("Text to Write\n");
}
```

C # WriteAllText ()

```
using System.IO;
using System.Text;
```

```
string filename = "c:\path\to\file.txt";
File.writeAllText(filename, "Text to write\n");
```

C # File.Exists ()

```
using System;
using System.IO;

public class Program
{
    public static void Main()
    {
        string filePath = "somePath";

        if(File.Exists(filePath))
        {
            Console.WriteLine("Exists");
        }
        else
        {
            Console.WriteLine("Does not exist");
        }
    }
}
```

Può essere utilizzato anche in un operatore ternario.

```
Console.WriteLine(File.Exists(pathToFile) ? "Exists" : "Does not exist");
```

Leggi File Input / Output online: https://riptutorial.com/it/dot-net/topic/1376/file-input---output

Capitolo 19: Gestione della memoria

Osservazioni

Le applicazioni critiche per le prestazioni nelle applicazioni .NET gestite possono essere gravemente influenzate dal GC. Quando viene eseguito il GC, tutti gli altri thread vengono sospesi fino al completamento. Per questo motivo, si consiglia di valutare attentamente i processi del GC e determinare come minimizzare quando viene eseguito.

Examples

Risorse non gestite

Quando parliamo di GC e di "heap", stiamo parlando di ciò che viene chiamato *heap gestito* . Gli oggetti *nell'heap gestito* possono accedere alle risorse non nell'heap gestito, ad esempio, durante la scrittura o la lettura da un file. Un comportamento imprevisto può verificarsi quando, un file viene aperto per la lettura e quindi si verifica un'eccezione, impedendo la chiusura dell'handle del file normalmente. Per questo motivo, .NET richiede che le risorse non gestite implementino l'interfaccia `IDisposable` . Questa interfaccia ha un unico metodo chiamato `Dispose` senza parametri:

```
public interface IDisposable
{
    Dispose();
}
```

Quando si gestiscono risorse non gestite, è necessario assicurarsi che siano correttamente smaltite. È possibile farlo chiamando esplicitamente `Dispose()` in un blocco `finally` o con un'istruzione `using` .

```
StreamReader sr;
string textFromFile;
string filename = "SomeFile.txt";
try
{
    sr = new StreamReader(filename);
    textFromFile = sr.ReadToEnd();
}
finally
{
    if (sr != null) sr.Dispose();
}
```

o

```
string textFromFile;
string filename = "SomeFile.txt";
```

```
using (StreamReader sr = new Streamreader(filename))
{
    textFromFile = sr.ReadToEnd();
}
```

Quest'ultimo è il metodo preferito e viene automaticamente esteso al primo durante la compilazione.

Usa SafeHandle durante il wrapping delle risorse non gestite

Quando si scrivono wrapper per risorse non gestite, è necessario `SafeHandle` sottoclasse `SafeHandle` piuttosto che provare a implementare `IDisposable` e un finalizzatore. `SafeHandle` sottoclasse `SafeHandle` dovrebbe essere il più piccola e semplice possibile per ridurre al minimo la possibilità di una perdita di handle. Ciò probabilmente significa che l'implementazione di SafeHandle fornirebbe un dettaglio di implementazione interna di una classe che lo avvolge per fornire un'API utilizzabile. Questa classe garantisce che, anche se un programma perde l'istanza di `SafeHandle` , l'handle non gestito viene rilasciato.

```
using System.Runtime.InteropServices;

class MyHandle : SafeHandle
{
    public override bool IsInvalid => handle == IntPtr.Zero;
    public MyHandle() : base(IntPtr.Zero, true)
    { }

    public MyHandle(int length) : this()
    {
        SetHandle(Marshal.AllocHGlobal(length));
    }

    protected override bool ReleaseHandle()
    {
        Marshal.FreeHGlobal(handle);
        return true;
    }
}
```

Dichiarazione di non responsabilità: questo esempio è un tentativo di mostrare come proteggere una risorsa gestita con `SafeHandle` che implementa `IDisposable` per te e configura i finalizzatori in modo appropriato. È molto inventato e probabilmente inutile allocare un blocco di memoria in questo modo.

Leggi Gestione della memoria online: https://riptutorial.com/it/dot-net/topic/59/gestione-della-memoria

Capitolo 20: Globalizzazione in ASP.NET MVC tramite l'internazionalizzazione intelligente per ASP.NET

Osservazioni

Internazionalizzazione intelligente per la pagina ASP.NET

Il vantaggio di questo approccio è che non è necessario ingombrare controller e altre classi con il codice per cercare valori dai file .resx. Si circonda semplicemente il testo in [[[parentesi quadre.]]] (Il delimitatore è configurabile.) Un `HttpModule` cerca una traduzione nel file .po per sostituire il testo delimitato. Se viene trovata una traduzione, `HttpModule` sostituisce la traduzione. Se non viene trovata alcuna traduzione, rimuove le parentesi quadre e rende la pagina con il testo originale non tradotto.

I file .po sono un formato standard per la fornitura di traduzioni per applicazioni, quindi sono disponibili numerose applicazioni per modificarle. È facile inviare un file .po a un utente non tecnico in modo che possano aggiungere traduzioni.

Examples

Configurazione di base e configurazione

1. Aggiungi il pacchetto nuget I18N al tuo progetto MVC.
2. In web.config, aggiungi `i18n.LocalizingModule` alla sezione `<httpModules>` o `<modules>` .

```
<!-- IIS 6 -->
<httpModules>
  <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
</httpModules>

<!-- IIS 7 -->
<system.webServer>
  <modules>
    <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
  </modules>
</system.webServer>
```

3. Aggiungi una cartella denominata "locale" alla radice del tuo sito. Crea una sottocartella per ogni cultura che desideri supportare. Ad esempio, `/locale/fr/` .
4. In ogni cartella specifica della cultura, crea un file di testo denominato `messages.po` .
5. A scopo di test, inserisci le seguenti righe di testo nel tuo file `messages.po` :

```
#: Translation test
msgid "Hello, world!"
```

```
msgstr "Bonjour le monde!"
```

6. Aggiungi un controller al tuo progetto che restituisce del testo da tradurre.

```
using System.Web.Mvc;

namespace I18nDemo.Controllers
{
    public class DefaultController : Controller
    {
        public ActionResult Index()
        {
            // Text inside [[[triple brackets]]] must precisely match
            // the msgid in your .po file.
            return Content("[[[Hello, world!]]]");
        }
    }
}
```

7. Esegui l'applicazione MVC e accedi al percorso corrispondente all'azione del controller, ad esempio http: // localhost: [yourportnumber] / default .
   Osserva che l'URL è cambiato per riflettere la tua cultura predefinita, come ad esempio http: // localhost: [yourportnumber] / it / default .
8. Sostituisci `/en/` nell'URL con `/fr/` (o qualsiasi cultura tu abbia selezionato.) La pagina ora dovrebbe visualizzare la versione tradotta del tuo testo.
9. Modifica le impostazioni della lingua del tuo browser per preferire la tua cultura alternativa e torna a `/default` nuovo. Osserva che l'URL viene modificato per riflettere la tua cultura alternativa e viene visualizzato il testo tradotto.
10. In web.config, aggiungi gestori in modo che gli utenti non possano navigare nella tua cartella `locale` .

```
<!-- IIS 6 -->
<system.web>
  <httpHandlers>
    <add path="*" verb="*" type="System.Web.HttpNotFoundHandler"/>
  </httpHandlers>
</system.web>

<!-- IIS 7 -->
<system.webServer>
  <handlers>
    <remove name="BlockViewHandler"/>
   <add name="BlockViewHandler" path="*" verb="*" preCondition="integratedMode"
type="System.Web.HttpNotFoundHandler"/>
  </handlers>
</system.webServer>
```

Leggi Globalizzazione in ASP.NET MVC tramite l'internazionalizzazione intelligente per ASP.NET online: https://riptutorial.com/it/dot-net/topic/5086/globalizzazione-in-asp-net-mvc-tramite-l-internazionalizzazione-intelligente-per-asp-net

Capitolo 21: Glossario degli acronimi

Examples

Acronimi Correlati .Net

Si noti che alcuni termini come JIT e GC sono abbastanza generici da poter essere applicati a molti ambienti di linguaggio di programmazione e runtime.

CLR: Common Language Runtime
IL: Intermediate Language
EE: Execution Engine
JIT: compilatore just-in-time
GC: Garbage Collector
OOM: memoria insufficiente
STA: appartamento a thread singolo
MTA: appartamento multi-thread

Leggi Glossario degli acronimi online: https://riptutorial.com/it/dot-net/topic/10939/glossario-degli-acronimi

Capitolo 22: Impostazione affinità processo e discussione

Parametri

Parametro	Dettagli
affinità	intero che descrive il set di processori su cui è consentito eseguire il processo. Ad esempio, su un sistema con 8 processori, se si desidera che il processo venga eseguito solo sui processori 3 e 4, si sceglie un'affinità come questa: 00001100 che equivale a 12

Osservazioni

L'affinità del processore di un thread è l'insieme di processori con cui ha una relazione. In altre parole, quelli su cui può essere programmato l'esecuzione.

L'affinità del processore rappresenta un po 'ogni processore. Il bit 0 rappresenta il processore uno, il bit 1 rappresenta il processore due e così via.

Examples

Ottieni la maschera di affinità del processo

```
    public static int GetProcessAffinityMask(string processName = null)
    {
        Process myProcess = GetProcessByName(ref processName);

        int processorAffinity = (int)myProcess.ProcessorAffinity;
        Console.WriteLine("Process {0} Affinity Mask is : {1}", processName,
FormatAffinity(processorAffinity));

        return processorAffinity;
    }

    public static Process GetProcessByName(ref string processName)
    {
        Process myProcess;
        if (string.IsNullOrEmpty(processName))
        {
            myProcess = Process.GetCurrentProcess();
            processName = myProcess.ProcessName;
        }
        else
        {
            Process[] processList = Process.GetProcessesByName(processName);
            myProcess = processList[0];
        }
```

```
        return myProcess;
    }

    private static string FormatAffinity(int affinity)
    {
        return Convert.ToString(affinity, 2).PadLeft(Environment.ProcessorCount, '0');
    }
}
```

Esempio di utilizzo:

```
private static void Main(string[] args)
{
    GetProcessAffinityMask();

    Console.ReadKey();
}
// Output:
// Process Test.vshost Affinity Mask is : 11111111
```

Imposta la maschera di affinità del processo

```
    public static void SetProcessAffinityMask(int affinity, string processName = null)
    {
        Process myProcess = GetProcessByName(ref processName);

        Console.WriteLine("Process {0} Old Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));

        myProcess.ProcessorAffinity = new IntPtr(affinity);
        Console.WriteLine("Process {0} New Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));
    }
```

Esempio di utilizzo:

```
private static void Main(string[] args)
{
    int newAffinity = Convert.ToInt32("10101010", 2);
    SetProcessAffinityMask(newAffinity);

    Console.ReadKey();
}
// Output :
// Process Test.vshost Old Affinity Mask is : 11111111
// Process Test.vshost New Affinity Mask is : 10101010
```

Leggi Impostazione affinità processo e discussione online: https://riptutorial.com/it/dot-net/topic/4431/impostazione-affinita-processo-e-discussione

Capitolo 23: impostazioni

Examples

AppSettings da ConfigurationSettings in .NET 1.x

Utilizzo sconsigliato

La classe ConfigurationSettings era il modo originale per recuperare le impostazioni per un assembly in .NET 1.0 e 1.1. È stato sostituito dalla classe ConfigurationManager e dalla classe WebConfigurationManager .

Se si hanno due chiavi con lo stesso nome nella sezione `appSettings` del file di configurazione, viene utilizzato l'ultimo.

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Program.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationSettings.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            string twoKeys = ConfigurationSettings.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

Leggere AppSettings da ConfigurationManager in .NET 2.0 e versioni successive

La classe ConfigurationManager supporta la proprietà `AppSettings` , che consente di continuare a leggere le impostazioni dalla sezione `appSettings` di un file di configurazione allo stesso modo di .NET 1.x supportato.

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Program.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationManager.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            var twoKeys = ConfigurationManager.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

Introduzione all'applicazione fortemente tipizzata e al supporto delle impostazioni utente di Visual Studio

Visual Studio aiuta a gestire le impostazioni dell'utente e dell'applicazione. L'utilizzo di questo approccio ha questi vantaggi rispetto all'utilizzo della sezione `appSettings` del file di configurazione.

1. Le impostazioni possono essere fatte fortemente battute. Qualsiasi tipo che può essere serializzato può essere utilizzato per un valore di impostazione.

2. Le impostazioni dell'applicazione possono essere facilmente separate dalle impostazioni dell'utente. Le impostazioni dell'applicazione sono memorizzate in un unico file di configurazione: `web.config` per siti Web e applicazioni Web e app.config, rinominato come *assembly* .exe.config, dove *assembly* è il nome dell'eseguibile. Le impostazioni utente (non utilizzate dai progetti Web) sono memorizzate in un file `user.config` nella cartella Dati dell'applicazione dell'utente (che varia con la versione del sistema operativo).

3. Le impostazioni dell'applicazione dalle librerie di classi possono essere combinate in un singolo file di configurazione senza il rischio di conflitti di nome, poiché ogni libreria di classi può avere una propria sezione delle impostazioni personalizzate.

Nella maggior parte dei tipi di progetto, la finestra di dialogo Proprietà progetto ha una scheda Impostazioni che è il punto di partenza per la creazione di applicazioni e impostazioni utente personalizzate. Inizialmente, la scheda Impostazioni sarà vuota, con un singolo collegamento per creare un file di impostazioni predefinito. Cliccando sul link si ottengono queste modifiche:

1. Se un file di configurazione (`app.config` o `web.config`) non esiste per il progetto, ne verrà creato uno.

2. La scheda Impostazioni verrà sostituita con un controllo griglia che consente di creare, modificare ed eliminare voci di impostazioni individuali.

3. In Esplora soluzioni, viene aggiunta una voce `Settings.settings` nella cartella speciale Proprietà. Aprendo questo elemento si aprirà la scheda Impostazioni.

4. Un nuovo file con una nuova classe parziale viene aggiunto nella cartella `Properties` nella cartella del progetto. Questo nuovo file è denominato `Settings.Designer.__` (.cs, .vb, ecc.) E la classe è denominata `Settings` . La classe è generata dal codice, quindi non dovrebbe essere modificata, ma la classe è una classe parziale, quindi è possibile estendere la classe inserendo membri aggiuntivi in un file separato. Inoltre, la classe viene implementata utilizzando Singleton Pattern, esponendo l'istanza singleton con la proprietà `Default` .

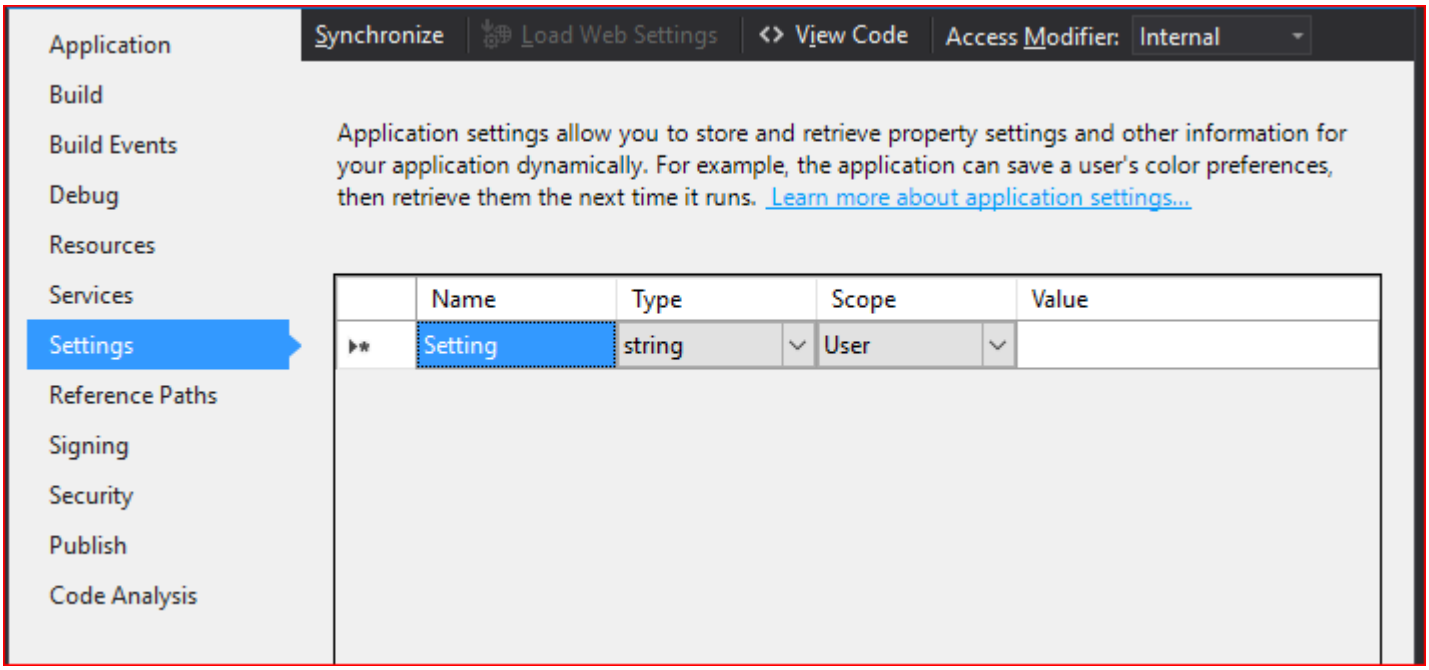
Quando aggiungi ogni nuova voce alla scheda Impostazioni, Visual Studio fa queste due cose:

1. Salva le impostazioni nel file di configurazione, in una sezione di configurazione personalizzata progettata per essere gestita dalla classe Impostazioni.

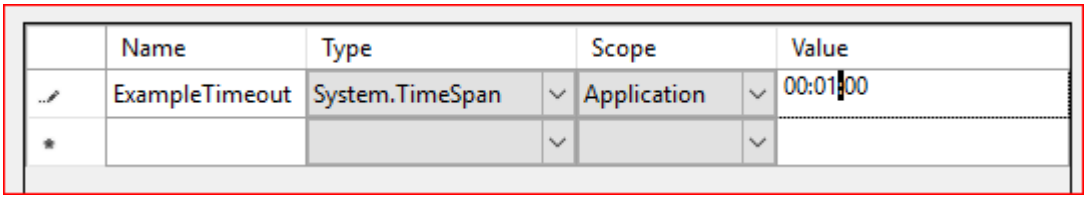
2. Crea un nuovo membro nella classe Impostazioni per leggere, scrivere e presentare le impostazioni nel tipo specifico selezionato dalla scheda Impostazioni.

Lettura delle impostazioni fortemente tipizzate dalla sezione personalizzata del file di configurazione

A partire da una nuova classe Settings e dalla sezione di configurazione personalizzata:



Aggiungere un'impostazione dell'applicazione denominata ExampleTimeout, utilizzando l'ora System.Timespan e impostare il valore su 1 minuto:



Salva le proprietà del progetto, che salva le voci della scheda Impostazioni, oltre a rigenerare la classe Impostazioni personalizzate e aggiorna il file di configurazione del progetto.

Usa l'impostazione dal codice (C #):

Program.cs

```
using System;
using System.Diagnostics;
using ConsoleApplication1.Properties;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            TimeSpan exampleTimeout = Settings.Default.ExampleTimeout;
            Debug.Assert(TimeSpan.FromMinutes(1).Equals(exampleTimeout));

            Console.ReadKey();
        }
    }
}
```

Sotto le coperte

Cerca nel file di configurazione del progetto per vedere come è stata creata la voce di impostazione dell'applicazione:

app.config (Visual Studio aggiorna automaticamente)

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="applicationSettings"
type="System.Configuration.ApplicationSettingsGroup, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" >
      <section name="ConsoleApplication1.Properties.Settings"
type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    </sectionGroup>
  </configSections>
  <appSettings />
  <applicationSettings>
    <ConsoleApplication1.Properties.Settings>
      <setting name="ExampleTimeout" serializeAs="String">
        <value>00:01:00</value>
      </setting>
    </ConsoleApplication1.Properties.Settings>
  </applicationSettings>
</configuration>
```

Si noti che la sezione `appSettings` non viene utilizzata. La sezione `applicationSettings` contiene una sezione personalizzata dello spazio dei nomi con un elemento di `setting` per ogni voce. Il tipo del valore non è memorizzato nel file di configurazione; è noto solo dalla classe `Settings` .

Cerca nella classe `Settings` per vedere come usa la classe `ConfigurationManager` per leggere questa sezione personalizzata.

Settings.designer.cs (per progetti C #)

```
...
    [global::System.Configuration.ApplicationScopedSettingAttribute()]
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.Configuration.DefaultSettingValueAttribute("00:01:00")]
    public global::System.TimeSpan ExampleTimeout {
        get {
            return ((global::System.TimeSpan)(this["ExampleTimeout"]));
        }
    }
...
```

Si noti che è stato creato un `DefaultSettingValueAttribute` per memorizzare il valore immesso nella scheda Impostazioni della finestra di dialogo Proprietà progetto. Se la voce non è presente nel file di configurazione, viene utilizzato questo valore predefinito.

Leggi impostazioni online: https://riptutorial.com/it/dot-net/topic/54/impostazioni

Capitolo 24: Iniezione di dipendenza

Osservazioni

Problemi risolti dall'iniezione delle dipendenze

Se non si utilizza l'iniezione di dipendenza, la classe `Greeter` potrebbe essere più simile a questa:

```
public class ControlFreakGreeter
{
    public void Greet()
    {
        var greetingProvider = new SqlGreetingProvider(
            ConfigurationManager.ConnectionStrings["myConnectionString"].ConnectionString);
        var greeting = greetingProvider.GetGreeting();
        Console.WriteLine(greeting);
    }
}
```

È un "maniaco del controllo" perché controlla la creazione della classe che fornisce il saluto, controlla da dove proviene la stringa di connessione SQL e controlla l'output.

Usando l'iniezione di dipendenza, la classe `Greeter` rinuncia a quelle responsabilità a favore di una singola responsabilità, scrivendo un saluto fornito ad essa.

Il principio di inversione di dipendenza suggerisce che le classi dovrebbero dipendere dalle astrazioni (come le interfacce) piuttosto che da altre classi concrete. Le dipendenze dirette (accoppiamento) tra classi possono rendere la manutenzione progressivamente difficile. A seconda delle astrazioni è possibile ridurre tale accoppiamento.

L'iniezione di dipendenza ci aiuta a raggiungere tale inversione di dipendenza perché conduce a scrivere classi che dipendono dalle astrazioni. La classe `Greeter` "non sa" nulla dei dettagli di implementazione di `IGreetingProvider` e `IGreetingWriter` . Sa solo che le dipendenze iniettate implementano quelle interfacce. Ciò significa che le modifiche alle classi concrete che implementano `IGreetingProvider` e `IGreetingWriter` non influiranno su `Greeter` . Né li sostituiranno con implementazioni completamente diverse. Verranno modificate solo le interfacce. `Greeter` è disaccoppiato.

`ControlFreakGreeter` è impossibile eseguire correttamente il test dell'unità. Vogliamo testare una piccola unità di codice, ma il nostro test includerebbe la connessione a SQL e l'esecuzione di una stored procedure. Includerebbe anche il test dell'output della console. Poiché ControlFreakGreeter fa così tanto è impossibile testare separatamente dalle altre classi.

`Greeter` è facile da testare in quanto è possibile iniettare implementazioni derise delle sue dipendenze che sono più facili da eseguire e verificare rispetto alla chiamata di una stored procedure o alla lettura dell'output della console. Non richiede una stringa di connessione in app.config.

Le implementazioni concrete di `IGreetingProvider` e `IGreetingWriter` potrebbero diventare più complesse. A loro volta, potrebbero avere le loro dipendenze che vengono iniettate in loro. (Ad esempio, inseriremmo la stringa di connessione SQL in `SqlGreetingProvider` .) Ma quella complessità è "nascosta" da altre classi che dipendono solo dalle interfacce. Ciò rende più semplice modificare una classe senza un "effetto a catena" che richiede di apportare modifiche corrispondenti ad altre classi.

Examples

Iniezione delle dipendenze - Semplice esempio

Questa classe si chiama `Greeter` . La sua responsabilità è di produrre un saluto. Ha due *dipendenze* . Ha bisogno di qualcosa che gli dia il benvenuto per l'output, e quindi ha bisogno di un modo per produrre quel saluto. Queste dipendenze sono entrambe descritte come interfacce, `IGreetingProvider` e `IGreetingWriter` . In questo esempio, queste due dipendenze vengono "iniettate" in `Greeter` . (Ulteriori spiegazioni sull'esempio.)

```
public class Greeter
{
    private readonly IGreetingProvider _greetingProvider;
    private readonly IGreetingWriter _greetingWriter;

    public Greeter(IGreetingProvider greetingProvider, IGreetingWriter greetingWriter)
    {
        _greetingProvider = greetingProvider;
        _greetingWriter = greetingWriter;
    }

    public void Greet()
    {
        var greeting = _greetingProvider.GetGreeting();
        _greetingWriter.WriteGreeting(greeting);
    }
}

public interface IGreetingProvider
{
    string GetGreeting();
}

public interface IGreetingWriter
{
    void WriteGreeting(string greeting);
}
```

La classe di `Greeting` dipende da `IGreetingProvider` e `IGreetingWriter` , ma non è responsabile della creazione di istanze di entrambi. Invece li richiede nel suo costruttore. Qualsiasi cosa crei un'istanza di `Greeting` deve fornire queste due dipendenze. Possiamo chiamarlo "iniettare" le dipendenze.

Poiché le dipendenze sono fornite alla classe nel suo costruttore, questa viene anche chiamata "iniezione costruttore".

Alcune convenzioni comuni:

- Il costruttore salva le dipendenze come campi `private` . Non appena la classe viene istanziata, tali dipendenze sono disponibili per tutti gli altri metodi non statici della classe.
- I campi `private` sono di `readonly` . Una volta impostati nel costruttore, non possono essere modificati. Questo indica che quei campi non dovrebbero (e non possono) essere modificati al di fuori del costruttore. Ciò garantisce inoltre che tali dipendenze siano disponibili per tutta la vita della classe.
- Le dipendenze sono interfacce. Questo non è strettamente necessario, ma è comune perché rende più facile sostituire un'implementazione della dipendenza con un'altra. Consente inoltre di fornire una versione mocked dell'interfaccia per scopi di test unitari.

In che modo l'iniezione delle dipendenze rende più semplice il test unitario

Questo si basa sul precedente esempio della classe `Greeter` che ha due dipendenze, `IGreetingProvider` e `IGreetingWriter` .

L'effettiva implementazione di `IGreetingProvider` potrebbe recuperare una stringa da una chiamata API o da un database. L'implementazione di `IGreetingWriter` potrebbe visualizzare il saluto nella console. Ma poiché `Greeter` ha le sue dipendenze iniettate nel suo costruttore, è facile scrivere un test unitario che inietta versioni derise di quelle interfacce. Nella vita reale potremmo usare un framework come Moq , ma in questo caso scriverò quelle implementazioni derise.

```
public class TestGreetingProvider : IGreetingProvider
{
    public const string TestGreeting = "Hello!";

    public string GetGreeting()
    {
        return TestGreeting;
    }
}

public class TestGreetingWriter : List<string>, IGreetingWriter
{
    public void WriteGreeting(string greeting)
    {
        Add(greeting);
    }
}

[TestClass]
public class GreeterTests
{
    [TestMethod]
    public void Greeter_WritesGreeting()
    {
        var greetingProvider = new TestGreetingProvider();
        var greetingWriter = new TestGreetingWriter();
        var greeter = new Greeter(greetingProvider, greetingWriter);
        greeter.Greet();
        Assert.AreEqual(greetingWriter[0], TestGreetingProvider.TestGreeting);
    }
}
```

Il comportamento di `IGreetingProvider` e `IGreetingWriter` non sono rilevanti per questo test. Vogliamo testare che `Greeter` riceve un saluto e lo scrive. Il design di `Greeter` (usando l'iniezione di dipendenza) ci permette di iniettare dipendenze derise senza parti mobili complicate. Tutto quello che stiamo testando è che `Greeter` interagisce con quelle dipendenze come ci aspettiamo.

Perché utilizziamo i contenitori di iniezione delle dipendenze (contenitori IoC)

Iniezione di dipendenza significa scrivere classi in modo che non controllino le loro dipendenze - invece, le loro dipendenze sono fornite a loro ("iniettato").

Questa non è la stessa cosa dell'utilizzo di un'infrastruttura per le dipendenze (spesso chiamata "contenitore DI", "contenitore IoC" o semplicemente "contenitore") come Castle Windsor, Autofac, SimpleInjector, Ninject, Unity o altri.

Un contenitore semplifica l'iniezione della dipendenza. Ad esempio, supponiamo di scrivere un numero di classi che si basano sull'integrazione delle dipendenze. Una classe dipende da diverse interfacce, le classi che implementano tali interfacce dipendono da altre interfacce e così via. Alcuni dipendono da valori specifici. E solo per divertimento, alcune di queste classi implementano `IDisposable` e devono essere smaltite.

Ogni singola classe è ben scritta e facile da testare. Ma ora c'è un altro problema: la creazione di un'istanza di una classe è diventata molto più complicata. Supponiamo di creare un'istanza di una classe `CustomerService` . Ha dipendenze e le sue dipendenze hanno dipendenze. La costruzione di un'istanza potrebbe essere simile a questa:

```
public CustomerData GetCustomerData(string customerNumber)
{
    var customerApiEndpoint =
ConfigurationManager.AppSettings["customerApi:customerApiEndpoint"];
    var logFilePath = ConfigurationManager.AppSettings["logwriter:logFilePath"];
    var authConnectionString =
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString;
    using(var logWriter = new LogWriter(logFilePath ))
    {
        using(var customerApiClient = new CustomerApiClient(customerApiEndpoint))
        {
            var customerService = new CustomerService(
                new SqlAuthorizationRepository(authorizationConnectionString, logWriter),
                new CustomerDataRepository(customerApiClient, logWriter),
                logWriter
            );

            // All this just to create an instance of CustomerService!
            return customerService.GetCustomerData(string customerNumber);
        }
    }
}
```

Potresti chiedertelo, perché non inserire l'intera costruzione gigante in una funzione separata che restituisce solo `CustomerService` ? Una ragione è che poiché le dipendenze di ogni classe vengono iniettate in essa, una classe non è responsabile per sapere se quelle dipendenze sono `IDisposable` o che le eliminano. Li usa solo. Quindi, se avessimo una funzione `GetCustomerService()` che

restituiva un `CustomerService` completamente costruito, quella classe potrebbe contenere un numero di risorse disponibili e nessun modo per accedervi o disporne.

E a parte il disporre di `IDisposable` , chi vuole chiamare una serie di costruttori annidati in questo modo, mai? Questo è un breve esempio. Potrebbe ottenere molto, molto peggio. Di nuovo, ciò non significa che abbiamo scritto le classi nel modo sbagliato. Le classi potrebbero essere individualmente perfette. La sfida è comporle insieme.

Un contenitore per l'iniezione di dipendenza lo semplifica. Ci permette di specificare quale classe o valore dovrebbe essere usato per soddisfare ogni dipendenza. Questo esempio un po 'semplicistico utilizza Castle Windsor:

```
var container = new WindsorContainer()
container.Register(
    Component.For<CustomerService>(),
    Component.For<ILogWriter, LogWriter>()
        .DependsOn(Dependency.OnAppSettingsValue("logFilePath", "logWriter:logFilePath")),
    Component.For<IAuthorizationRepository, SqlAuthorizationRepository>()
        .DependsOn(Dependency.OnValue(connectionString,
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString)),
    Component.For<ICustomerDataProvider, CustomerApiClient>()
         .DependsOn(Dependency.OnAppSettingsValue("apiEndpoint",
"customerApi:customerApiEndpoint"))
);
```

Chiamiamo questo "registrazione delle dipendenze" o "configurazione del contenitore". Tradotto, questo dice al nostro `WindsorContainer` :

- Se una classe richiede `ILogWriter` , creare un'istanza di `LogWriter` . `LogWriter` richiede un percorso file. Utilizzare questo valore da `AppSettings` .
- Se una classe richiede `IAuthorizationRepository` , creare un'istanza di `SqlAuthorizationRepository` . Richiede una stringa di connessione. Utilizzare questo valore dalla sezione `ConnectionStrings` .
- Se una classe richiede `ICustomerDataProvider` , creare un `CustomerApiClient` e fornire la stringa necessaria da `AppSettings` .

Quando chiediamo una dipendenza dal contenitore, chiamiamo "risolvere" una dipendenza. È una cattiva pratica farlo direttamente utilizzando il contenitore, ma questa è una storia diversa. A scopo dimostrativo, ora potremmo fare questo:

```
var customerService = container.Resolve<CustomerService>();
var data = customerService.GetCustomerData(customerNumber);
container.Release(customerService);
```

Il contenitore sa che `CustomerService` dipende da `IAuthorizationRepository` e `ICustomerDataProvider` . Sa quali classi deve creare per soddisfare tali requisiti. Queste classi, a loro volta, hanno più dipendenze e il contenitore sa come soddisfarle. Creerà ogni classe di cui ha bisogno fino a quando non potrà restituire un'istanza di `CustomerService` .

Se arriva a un punto in cui una classe richiede una dipendenza che non abbiamo registrato, come `IDoesSomethingElse` , quando proviamo a risolvere `CustomerService` genererà un'eccezione chiara

che ci dice che non abbiamo registrato nulla per soddisfare tale requisito.

Ogni framework DI si comporta in modo leggermente diverso, ma in genere ci dà il controllo sul modo in cui determinate classi vengono istanziate. Ad esempio, vogliamo creare un'istanza di `LogWriter` e fornirla ad ogni classe che dipende da `ILogWriter` , oppure vogliamo che ne crei una nuova ogni volta? La maggior parte dei contenitori ha un modo per specificarlo.

Che dire delle classi che implementano `IDisposable` ? Ecco perché chiamiamo `container.Release(customerService);` alla fine. La maggior parte dei contenitori (incluso Windsor) eseguirà un passo indietro attraverso tutte le dipendenze create e `Dispose` quelle che devono essere eliminate. Se `CustomerService` è `IDisposable` lo `IDisposable` anche.

La registrazione delle dipendenze come visto sopra potrebbe sembrare un po 'più codice da scrivere. Ma quando abbiamo un sacco di classi con un sacco di dipendenze, allora paga davvero. E se dovessimo scrivere quelle stesse classi *senza* usare l'injection dependency, quella stessa applicazione con molte classi diventerebbe difficile da mantenere e testare.

Questo graffia la superficie del *motivo per* cui utilizziamo i contenitori di iniezione di dipendenza. *Il modo in* cui configuriamo la nostra applicazione per usarne uno (e usarlo correttamente) non è solo un argomento: è un numero di argomenti, in quanto le istruzioni e gli esempi variano da un contenitore all'altro.

Leggi Iniezione di dipendenza online: https://riptutorial.com/it/dot-net/topic/5085/iniezione-di-dipendenza

Capitolo 25: JSON in .NET con Newtonsoft.Json

introduzione

Il pacchetto `Newtonsoft.Json` è diventato lo standard defacto per l'utilizzo e la manipolazione di testo e oggetti con formattazione JSON in .NET. È uno strumento robusto veloce e facile da usare.

Examples

Serializza l'oggetto in JSON

```
using Newtonsoft.Json;

var obj = new Person
{
    Name = "Joe Smith",
    Age = 21
};
var serializedJson = JsonConvert.SerializeObject(obj);
```

Questo risulta in questo JSON: `{"Name":"Joe Smith","Age":21}`

Deserializzare un oggetto dal testo JSON

```
var json = "{\"Name\":\"Joe Smith\",\"Age\":21}";
var person = JsonConvert.DeserializeObject<Person>(json);
```

Questo produce un oggetto `Person` con Nome "Joe Smith" e 21 anni.

Leggi JSON in .NET con Newtonsoft.Json online: https://riptutorial.com/it/dot-net/topic/8746/json-in--net-con-newtonsoft-json

Capitolo 26: Lavora con SHA1 in C

introduzione

in questo progetto vedi come lavorare con la funzione hash crittografica SHA1. per esempio ottieni hash dalla stringa e come decifrare l'hash SHA1. fonte su git hub:
https://github.com/mahdiabasi/SHA1Tool

Examples

#Controllare il checksum SHA1 di una funzione di file

Innanzitutto aggiungi System.Security.Cryptography e System.IO al tuo progetto

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

Leggi Lavora con SHA1 in C # online: https://riptutorial.com/it/dot-net/topic/9457/lavora-con-sha1-in-c-sharp

Capitolo 27: Lavora con SHA1 in C

introduzione

in questo progetto vedi come lavorare con la funzione hash crittografica SHA1. per esempio ottieni hash dalla stringa e come decifrare l'hash SHA1.

fonte compelete su github: https://github.com/mahdiabasi/SHA1Tool

Examples

#Controllare il checksum SHA1 di un file

> per prima cosa aggiungi lo spazio dei nomi System.Security.Cryptography al tuo progetto

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

#Generare hash di un testo

```
  public static string TextToHash(string text)
    {
        var sh = SHA1.Create();
        var hash = new StringBuilder();
        byte[] bytes = Encoding.UTF8.GetBytes(text);
        byte[] b = sh.ComputeHash(bytes);
        foreach (byte a in b)
        {
            var h = a.ToString("x2");
            hash.Append(h);
        }
        return hash.ToString();
    }
```

Leggi Lavora con SHA1 in C # online: https://riptutorial.com/it/dot-net/topic/9458/lavora-con-sha1-in-c-sharp

Capitolo 28: Lettura e scrittura di file zip

introduzione

La classe **ZipFile** risiede nello spazio dei nomi **System.IO.Compression** . Può essere utilizzato per leggere e scrivere su file Zip.

Osservazioni

- Puoi anche usare un MemoryStream invece di un FileStream.
- eccezioni

Eccezione	Condizione
ArgumentException	Lo stream è già stato chiuso o le funzionalità dello stream non corrispondono alla modalità (ad esempio: provare a scrivere su un flusso di sola lettura)
ArgumentNullException	il *flusso di* input è nullo
ArgumentOutOfRangeException	*la modalità* ha un valore non valido
InvalidDataException	Vedi la lista qui sotto

Quando viene lanciata una **InvalidDataException** , può avere 3 cause:

- Il contenuto dello stream non può essere interpretato come un archivio zip
- *la modalità* è Aggiorna e manca una voce dall'archivio o è corrotta e non può essere letta
- *la modalità* è Aggiorna e una voce è troppo grande per adattarsi alla memoria

Tutte le informazioni sono state prese da questa pagina MSDN

Examples

Elenco dei contenuti ZIP

Questo frammento elencherà tutti i nomi di file di un archivio zip. I nomi dei file sono relativi alla radice zip.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    for (int i = 0; i < archive.Entries.Count; i++)
    {
        Console.WriteLine($"{i}: {archive.Entries[i]}");
```

```
    }
}
```

Estrazione di file da file ZIP

Estrarre tutti i file in una directory è molto semplice:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    archive.ExtractToDirectory(AppDomain.CurrentDomain.BaseDirectory);
}
```

Quando il file esiste già, verrà lanciata una **System.IO.IOException** .

Estrazione di file specifici:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    // Get a root entry file
    archive.GetEntry("test.txt").ExtractToFile("test_extracted_getentries.txt", true);

    // Enter a path if you want to extract files from a subdirectory
    archive.GetEntry("sub/subtest.txt").ExtractToFile("test_sub.txt", true);

    // You can also use the Entries property to find files
    archive.Entries.FirstOrDefault(f => f.Name ==
"test.txt")?.ExtractToFile("test_extracted_linq.txt", true);

    // This will throw a System.ArgumentNullException because the file cannot be found
    archive.GetEntry("nonexistingfile.txt").ExtractToFile("fail.txt", true);
}
```

Ognuno di questi metodi produrrà lo stesso risultato.

Aggiornamento di un file ZIP

Per aggiornare un file ZIP, il file deve essere aperto con ZipArchiveMode.Update.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Update))
{
    // Add file to root
    archive.CreateEntryFromFile("test.txt", "test.txt");

    // Add file to subfolder
    archive.CreateEntryFromFile("test.txt", "symbols/test.txt");
}
```

C'è anche la possibilità di scrivere direttamente in un file all'interno dell'archivio:

```
var entry = archive.CreateEntry("createentry.txt");
```

```
using(var writer = new StreamWriter(entry.Open()))
{
    writer.WriteLine("Test line");
}
```

Leggi Lettura e scrittura di file zip online: https://riptutorial.com/it/dot-net/topic/9943/lettura-e-scrittura-di-file-zip

Capitolo 29: LINQ

introduzione

LINQ (Language Integrated Query) è un'espressione che recupera i dati da un'origine dati. LINQ semplifica questa situazione offrendo un modello coerente per lavorare con i dati attraverso vari tipi di fonti e formati di dati. In una query LINQ, si lavora sempre con gli oggetti. 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 un provider è disponibile. LINQ può essere utilizzato in C # e VB.

Sintassi

- public static TSource Aggregate <TSource> (questa sorgente IEnumerable <TSource>, Func <TSource, TSource, TSource> func)
- TAccumulate Aggregate statico pubblico <TSource, TAccumulate> (questa sorgente IEnumerable <TSource>, seme TAccumulate, Func <TAccumulate, TSource, TAccumulate> func)
- statico pubblico TResult Aggregate <TSource, TAccumulate, TResult> (questa sorgente IEnumerable <TSource>, TAccumulate seed, Func <TAccumulate, TSource, TAccumulate>, Func <TAccumulate, TResult> resultSelector)
- public static Booleano Tutti <TSource> (questo predicato IEnumerable <Origine TSource, Func <TSource, Boolean>)
- public static Boolean Any <TSource> (questa origine IEnumerable <TSource>)
- public static Booleano Any <TSource> (questo predicato IEnumerable <Origine TS, Fun <TSource, Boolean>)
- public static IEnumerable <TSource> AsEnumerable <TSource> (questa sorgente IEnumerable <TSource>)
- Media decimale statica pubblica (questa origine IEnumerable <Decimal>)
- Double Average statico pubblico (questo IEnumerable <Double> origine)
- Double Average statico pubblico (questa sorgente IEnumerable <Int32>)
- Double Average statico pubblico (questa sorgente IEnumerable <Int64>)
- public static Nullable <Decimal> Average (questa fonte IEnumerable <Nullable <Decimal >>)
- public static Nullable <Double> Average (this IEnumerable <Nullable <Double >> source)
- public static Nullable <Double> Average (questa sorgente IEnumerable <Nullable <Int32 >>)
- public static Nullable <Double> Average (questa fonte IEnumerable <Nullable <Int64 >>)
- public static Nullable <Single> Average (this IEnumerable <Nullable <Single >> source)
- Media statica pubblica singola (questa origine IEnumerable <Single>)
- Media decimale statica media <TSource> (questo selettore IEnumerable <TSource>, Func <TSource, Decimal>)
- media statica Double Average <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Double>)
- media statica Double Average <TSource> (questo componente IEnumerable <TSource>, Func <TSource, Int32>)

- media statica Double Average <TSource> (questo selettore IEnumerable <TSource>, Func <TSource, Int64>)
- public static Nullable <Decimal> Average <TSource> (questo IEnumerable <Origine TSource, Func <TSource, Nullable <Decimale >> Selettore)
- public static Nullable <Double> Average <TSource> (questa sorgente IEnumerable <TSource>, Func <TSource, Nullable <Double >>)
- public static Nullable <Double> Average <TSource> (questo IEnumerable <Origine TSource> Func <TSource, Nullable <Int32 >> selector)
- public static Nullable <Double> Average <TSource> (questo IEnumerable <Origine TSource> Func <TSource, Nullable <Int64 >> selector)
- public static Nullable <Single> Average <TSource> (questa sorgente IEnumerable <TSource>, Func <TSource, Nullable <Single >>)
- media singola statica pubblica <TSource> (questo selettore IEnumerable <Origine TSource, Func <TSource, Single>)
- public static IEnumerable <TResult> Cast <TResult> (questa sorgente IEnumerable)
- public static IEnumerable <TSource> Concat <TSource> (questo IEnumerable <TSource> first, IEnumerable <TSource> secondo)
- public static Boolean Contiene <TSource> (questa origine IEnumerable <TSource>, valore TSource)
- public static Boolean Contiene <TSource> (questa origine IEnumerable <TSource>, valore TSource, Comparatore IEqualityComparer <TSource>)
- Conteggio Int32 statico pubblico <TSource> (questa origine IEnumerable <TSource>)
- Conteggio Int32 statico pubblico <TSource> (questo predicato IEnumerable <Origine TSource, Func <TSource, Boolean>)
- public static IEnumerable <TSource> DefaultIfEmpty <TSource> (questa origine IEnumerable <TSource>)
- public static IEnumerable <TSource> DefaultIfEmpty <TSource> (questa sorgente IEnumerable <TSource>, TSource defaultValue)
- public static IEnumerable <TSource> Distinct <TSource> (questa origine IEnumerable <TSource>)
- public static IEnumerable <TSource> Distinct <TSource> (questa sorgente IEnumerable <TSource>, IEqualityComparer <TSource> comparer)
- public static TSource ElementAt <TSource> (questa sorgente IEnumerable <TSource>, indice Int32)
- pubblico statico TSource ElementAtOrDefault <TSource> (questa origine IEnumerable <TSource>, indice Int32)
- public static IEnumerable <TResult> Empty <TResult> ()
- public static IEnumerable <TSource> Eccetto <TSource> (questo IEnumerable <TSource> first, IEnumerable <TSource> secondo)
- public static IEnumerable <TSource> Tranne <TSource> (questo IEnumerable <TSource> first, IEnumerable <TSource> secondo, IEqualityComparer <TSource> comparer)
- public static TSource First <TSource> (questa origine IEnumerable <TSource>)
- public static TSource First <TSource> (questo predicato IEnumerable <TSource> source, Func <TSource, Boolean>)
- public static TSource FirstOrDefault <TSource> (questa origine IEnumerable <TSource>)
- public static TSource FirstOrDefault <TSource> (questo predicato IEnumerable <TSource>

source, Func <TSource, Boolean>)

- public static IEnumerable <IGrouping <TKey, TSource >> GroupBy <TSource, TKey> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IEnumerable <IGrouping <TKey, TSource >> GroupBy <TSource, TKey> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> comparer)
- public static IEnumerable <IGrouping <TKey, TElement >> GroupBy <TSource, TKey, TElement> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- public static IEnumerable <IGrouping <TKey, TElement >> GroupBy <TSource, TKey, TElement> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> comparer)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TResult> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TKey, IEnumerable <TSource>, TResult> resultSelector)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TResult> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TKey, IEnumerable <TSource>, TResult> resultSelector, IEqualityComparer <TKey> comparer)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TElement, TResult> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, Func <TKey, IEnumerable <TElement>, TResult > resultSelector)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TElement, TResult> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, Func <TKey, IEnumerable <TElement>, TResult > resultSelector, IEqualityComparer <TKey> comparer)
- public static IEnumerable <TResult> GroupJoin <TOuter, TInner, TKey, TResult> (questo IEnumerable <TOuter> esterno, IEnumerable <TInner> interno, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, IEnumerable <TInner>, TResult> resultSelector)
- public static IEnumerable <TResult> GroupJoin <TOuter, TInner, TKey, TResult> (questo IEnumerable <TOuter> esterno, IEnumerable <TInner> interno, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, IEnumerable <TInner>, TResult> resultSelector, IEqualityComparer <TKey> comparer)
- public static IEnumerable <TSource> Intersect <TSource> (questo IEnumerable <TSource> first, IEnumerable <TSource> second)
- public static IEnumerable <TSource> Intersect <TSource> (questo IEnumerable <TSource> first, IEnumerable <TSource> secondo, IEqualityComparer <TSource> comparer)
- public static IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> (questo IEnumerable <TOuter> esterno, IEnumerable <TInner> interno, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, TInner, TResult> resultSelector)
- public static IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> (questo IEnumerable <TOuter> esterno, IEnumerable <TInner> interno, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, TInner, TResult> resultSelector, IEqualityComparer <TKey> comparer)

- public static TSource Last <TSource> (questa fonte IEnumerable <TSource>)
- public static TSource Last <TSource> (questo predicato IEnumerable <TSource> source, Func <TSource, Boolean>)
- public static TSource LastOrDefault <TSource> (questa origine IEnumerable <TSource>)
- public static TSource LastOrDefault <TSource> (questo predicato IEnumerable <TSource>, Func <TSource, Boolean>)
- public static Int64 LongCount <TSource> (questa origine IEnumerable <TSource>)
- public static Int64 LongCount <TSource> (questo predicato IEnumerable <TSource> source, Func <TSource, Boolean>)
- Max decimale statico pubblico (questa origine IEnumerable <Decimal>)
- Double Max statico pubblico (questa sorgente <Double> di IEnumerable)
- Int32 Max statico pubblico (questa origine IEnumerable <Int32>)
- pubblico statico Int64 Max (questa sorgente IEnumerable <Int64>)
- public static Nullable <Decimal> Max (questa sorgente IEnumerable <Nullable <Decimal >>)
- public static Nullable <Double> Max (questa sorgente IEnumerable <Nullable <Double >>)
- statico pubblico Nullable <Int32> Max (questa sorgente IEnumerable <Nullable <Int32 >>)
- public static Nullable <Int64> Max (questa sorgente IEnumerable <Nullable <Int64 >>)
- public static Nullable <Single> Max (questa sorgente IEnumerable <Nullable <Single >>)
- Single Max pubblico statico (questa sorgente IEnumerable <Single>)
- public static TSource Max <TSource> (questa origine IEnumerable <TSource>)
- decimale statico pubblico massimo <TSource> (questo selettore IEnumerable <TSource>, Func <TSource, Decimal>)
- public static Double Max <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Double>)
- public static Int32 Max <TSource> (questo selettore IEnumerable <TSource>, Func <TSource, Int32>)
- public static Int64 Max <TSource> (questo selettore IEnumerable <TSource>, Func <TSource, Int64>)
- public static Nullable <Decimal> Max <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Nullable <Decimal >>)
- public static Nullable <Double> Max <TSource> (questa sorgente IEnumerable <TSource>, Func <TSource, Nullable <Double >>)
- public static Nullable <Int32> Max <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Nullable <Int32 >>)
- public static Nullable <Int64> Max <TSource> (questo IEnumerable <Origine TSource> Func <TSource, Nullable <Int64 >> selector)
- public static Nullable <Single> Max <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Nullable <Single >>)
- public static Single Max <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Single>)
- pubblico statico TResult Max <TSource, TResult> (questo sorgente IEnumerable <TSource>, Func <TSource, TResult>)
- Minimo statico pubblico Minimo (questa origine IEnumerable <Decimal>)
- Double Min statico pubblico (questa origine IEnumerable <Double>)
- public static Int32 Min (questa sorgente IEnumerable <Int32>)
- pubblico statico Int64 Min (questa sorgente IEnumerable <Int64>)

- public static Nullable <Decimal> Min (questa fonte IEnumerable <Nullable <Decimal >>)
- public static Nullable <Double> Min (questa sorgente IEnumerable <Nullable <Double >>)
- public static Nullable <Int32> Min (questa sorgente IEnumerable <Nullable <Int32 >>)
- public static Nullable <Int64> Min (questa sorgente IEnumerable <Nullable <Int64 >>)
- statico pubblico Nullable <Singolo> Min (questo oggetto IEnumerable <Nullable <Single >>)
- Single Min statico pubblico (questa origine IEnumerable <Single>)
- public static TSource Min <TSource> (questa origine IEnumerable <TSource>)
- public static Decimal Min <TSource> (questo selettore IEnumerable <TSource>, Func <TSource, Decimal>)
- public static Double Min <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Double>)
- public static Int32 Min <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Int32>)
- public static Int64 Min <TSource> (questo selettore IEnumerable <TSource>, Func <TSource, Int64>)
- public static Nullable <Decimal> Min <TSource> (questo sorgente IEnumerable <Origine TS, Func <TSource, Nullable <Decimale >>)
- public static Nullable <Double> Min <TSource> (questa sorgente IEnumerable <TSource>, Func <TSource, Nullable <Double >>)
- public static Nullable <Int32> Min <TSource> (questo sorgente IEnumerable <Origine TS, Func <TSource, Nullable <Int32 >>)
- public static Nullable <Int64> Min <TSource> (questo IEnumerable <Origine TSource> Func <TSource, Nullable <Int64 >> selector)
- public static Nullable <Single> Min <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Nullable <Single >>)
- public static Single Min <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Single>)
- pubblico statico TResult Min <TSource, TResult> (questo sorgente IEnumerable <TSource>, Func <TSource, TResult>)
- public static IEnumerable <TResult> OfType <TResult> (questa sorgente IEnumerable)
- public static IOrderedEnumerable <TSource> OrderBy <TSource, TKey> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> OrderBy <TSource, TKey> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> comparer)
- public static IOrderedEnumerable <TSource> OrderByDescending <TSource, TKey> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> OrderByDescending <TSource, TKey> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> comparer)
- Intervallo pubblico IEnumerable <Int32> public (Int32 start, Int32 count)
- public static IEnumerable <TResult> Repeat <TResult> (elemento TResult, numero Int32)
- public static IEnumerable <TSource> Reverse <TSource> (questa sorgente IEnumerable <TSource>)
- public static IEnumerable <TResult> Seleziona <TSource, TResult> (questo sorgente IEnumerable <TSource>, Func <TSource, TResult>)

- public static IEnumerable <TResult> Seleziona <TSource, TResult> (questo sorgente IEnumerable <TSource>, Func <TSource, Int32, TResult>)
- public static IEnumerable <TResult> SelectMany <TSource, TResult> (questo sorgente IEnumerable <TSource>, Func <TSource, IEnumerable <TResult >>)
- public static IEnumerable <TResult> SelectMany <TSource, TResult> (questo sorgente IEnumerable <TSource>, Func <TSource, Int32, IEnumerable <TResult >>)
- public static IEnumerable <TResult> SelectMany <TSource, TCollection, TResult> (questa sorgente IEnumerable <TSource>, Func <TSource, IEnumerable <TCollection >> collectionSelector, Func <TSource, TCollection, TResult> resultSelector)
- public static IEnumerable <TResult> SelectMany <TSource, TCollection, TResult> (questa sorgente IEnumerable <TSource>, Func <TSource, Int32, IEnumerable <TCollection >> collectionSelector, Func <TSource, TCollection, TResult> resultSelector)
- public static Boolean SequenceEqual <TSource> (questo IEnumerable <TSource> first, IEnumerable <TSource> second)
- public static Boolean SequenceEqual <TSource> (questo IEnumerable <TSource> first, IEnumerable <TSource> secondo, IEqualityComparer <TSource> comparer)
- public static TSource Single <TSource> (questa sorgente IEnumerable <TSource>)
- public static TSource Single <TSource> (questo predicato IEnumerable <TSource> source, Func <TSource, Boolean>)
- public static TSource SingleOrDefault <TSource> (questa origine IEnumerable <TSource>)
- public static TSource SingleOrDefault <TSource> (questo predicato IEnumerable <TSource> source, Func <TSource, Boolean>)
- public static IEnumerable <TSource> Salta <TSource> (questa sorgente IEnumerable <TSource>, conteggio Int32)
- public static IEnumerable <TSource> SkipWhile <TSource> (questo predicato IEnumerable <TSource>, Func <TSource, Boolean>)
- statico pubblico IEnumerable <TSource> SkipWhile <TSource> (questo predicato IEnumerable <TSource>, Func <TSource, Int32, Boolean>)
- Somma decimale statica pubblica (questa origine IEnumerable <Decimal>)
- Double Sum statico pubblico (questa origine <Double> di IEnumerable)
- Somma Int32 statica pubblica (questa origine IEnumerable <Int32>)
- Somma Int64 statica pubblica (questa origine IEnumerable <Int64>)
- Sommario <decimale> Nullable statico pubblico (questa origine IEnumerable <Nullable <Decimal >>)
- public static Nullable <Double> Sum (this IEnumerable <Nullable <Double >> source)
- Somma statica pubblica Nullable <Int32> (questa origine IEnumerable <Nullable <Int32 >>)
- Somma statica pubblica Nullable <Int64> (questa origine IEnumerable <Nullable <Int64 >>)
- public static Nullable <Single> Sum (this IEnumerable <Nullable <Single >> source)
- Somma statica pubblica singola (questa origine IEnumerable <Single>)
- Somma decimale statica pubblica <TSource> (questo selettore IEnumerable <TSource>, Func <TSource, Decimal>)
- Double Sum statico pubblico <TSource> (questo sorgente IEnumerable <Origine TSource, Func <TSource, Double>)
- Somma Intica statica pubblica <TSource> (questo sorgente IEnumerable <Origine TSource, Func <TSource, Int32>)
- Somma Int64 statica pubblica <TSource> (questo selettore IEnumerable <TSource>, Func

<TSource, Int64>)
- public static Nullable <Decimal> Sum <TSource> (questo IEnumerable <Origine TSource, Func <TSource, Nullable <Decimale >> Selettore)
- public static Nullable <Double> Sum <TSource> (questa sorgente IEnumerable <TSource>, Func <TSource, Nullable <Double >>)
- public static Nullable <Int32> Sum <TSource> (questo IEnumerable <Origine TSource> Func <TSource, Nullable <Int32 >> selector)
- public static Nullable <Int64> Sum <TSource> (questo componente IEnumerable <TSource>, Func <TSource, Nullable <Int64 >>)
- public static Nullable <Single> Sum <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Nullable <Single >>)
- Somma statica pubblica singola <TSource> (questo selettore IEnumerable <TSource>, Func <TSource, Single>)
- public static IEnumerable <TSource> Take <TSource> (questa sorgente IEnumerable <TSource>, conteggio Int32)
- public static IEnumerable <TSource> TakeWhile <TSource> (questo predicato IEnumerable <TSource> source, Func <TSource, Boolean>)
- public static IEnumerable <TSource> TakeWhile <TSource> (questo predicato IEnumerable <TSource>, Func <TSource, Int32, Boolean>)
- public static IOrderedEnumerable <TSource> ThenBy <TSource, TKey> (questa sorgente IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> ThenBy <TSource, TKey> (questa sorgente IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> comparer)
- public static IOrderedEnumerable <TSource> ThenByDescending <TSource, TKey> (questa sorgente IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> ThenByDescending <TSource, TKey> (questa sorgente IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> comparer)
- public static TSource [] ToArray <TSource> (questa sorgente IEnumerable <TSource>)
- Dizionario statico pubblico <TKey, TSource> ToDictionary <TSource, TKey> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- Dizionario statico pubblico <TKey, TSource> ToDictionary <TSource, TKey> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> comparer)
- Dizionario statico pubblico <TKey, TElement> ToDictionary <TSource, TKey, TElement> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- Dizionario statico pubblico <TKey, TElement> ToDictionary <TSource, TKey, TElement> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> comparer)
- elenco statico pubblico <TSource> ToList <TSource> (questa origine IEnumerable <TSource>)
- public static ILookup <TKey, TSource> ToLookup <TSource, TKey> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static ILookup <TKey, TSource> ToLookup <TSource, TKey> (questa sorgente

IEnumerable <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> comparer)
- public static ILookup <TKey, TElement> ToLookup <TSource, TKey, TElement> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- public static ILookup <TKey, TElement> ToLookup <TSource, TKey, TElement> (questa sorgente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> comparer)
- public static IEnumerable <TSource> Union <TSource> (questo IEnumerable <TSource> first, IEnumerable <TSource> second)
- public static IEnumerable <TSource> Union <TSource> (questo IEnumerable <TSource> first, IEnumerable <TSource> secondo, IEqualityComparer <TSource> comparer)
- public static IEnumerable <TSource> Dove <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Boolean>)
- public static IEnumerable <TSource> Dove <TSource> (questo sorgente IEnumerable <TSource>, Func <TSource, Int32, Boolean>)
- public static IEnumerable <TResult> Zip <TFirst, TSecond, TResult> (questo IEnumerable <TFirst> primo, IEnumerable <TSecond> secondo, Func <TFirst, TSecond, TResult> resultSelector)

Osservazioni

- Vedi anche LINQ .

I metodi integrati LINQ sono metodi di estensione per l'interfaccia `IEnumerable<T>` che risiedono nella classe `System.Linq.Enumerable` nell'assembly `System.Core` . Sono disponibili in .NET Framework 3.5 e versioni successive.

LINQ consente la modifica, la trasformazione e la combinazione di vari `IEnumerable` usando una sintassi di tipo query o funzionale.

Mentre i metodi LINQ standard possono funzionare su qualsiasi `IEnumerable<T>` , compresi gli array semplici e `List<T>` s, possono essere utilizzati anche su oggetti di database, in cui l'insieme di espressioni LINQ può essere trasformato in molti casi in SQL se il l'oggetto dati lo supporta. Vedi LINQ to SQL .

Per i metodi che confrontano gli oggetti (come `Contains` ed `Except`), `IEquatable<T>.Equals` viene utilizzato se il tipo T della raccolta implementa tale interfaccia. In caso contrario, vengono utilizzati gli standard `Equals` e `GetHashCode` del tipo (eventualmente sovrascritti dalle implementazioni `Object` predefinite). Esistono anche sovraccarichi per questi metodi che consentono di specificare un `IEqualityComparer<T>` personalizzato `IEqualityComparer<T>` .

Per i metodi `...OrDefault` , il `default(T)` viene utilizzato per generare valori predefiniti.

Riferimento ufficiale: classe enumerabile

Valutazione pigra

Praticamente ogni query che restituisce un `IEnumerable<T>` non viene valutata immediatamente; invece, la logica viene ritardata fino a quando la query non viene ripetuta. Una implicazione è che ogni volta che qualcuno esegue un'iterazione su un oggetto `IEnumerable<T>` creato da una di queste query, ad esempio, `.Where()` , viene ripetuta la logica di query completa. Se il predicato è di lunga durata, ciò può essere causa di problemi di prestazioni.

Una soluzione semplice (quando si conosce o può controllare la dimensione approssimativa della sequenza risultante) consiste nel bufferizzare completamente i risultati utilizzando `.ToArray()` o `.ToList()` . `.ToDictionary()` o `.ToLookup()` può svolgere lo stesso ruolo. Si può anche, ovviamente, iterare sull'intera sequenza e bufferizzare gli elementi secondo un'altra logica personalizzata.

`ToArray()` o `ToList()` ?

Sia `.ToArray()` che `.ToList()` ciclo di tutti gli elementi di una sequenza I `IEnumerable<T>` `.ToList()` `IEnumerable<T>` e salvano i risultati in una raccolta archiviata in memoria. Utilizzare le seguenti linee guida per determinare quale scegliere:

- Alcune API potrebbero richiedere un `T[]` o un `List<T>` .
- `.ToList()` genere gira più velocemente e genera meno garbage di `.ToArray()` , perché quest'ultimo deve copiare tutti gli elementi in una nuova raccolta di dimensioni fisse ancora una volta rispetto al primo, in quasi tutti i casi.
- Gli elementi possono essere aggiunti o rimossi dall'elenco `List<T>` restituito da `.ToList()` , mentre il `T[]` restituito da `.ToArray()` rimane una dimensione fissa per tutta la sua durata. In altre parole, `List<T>` è mutabile e `T[]` è immutabile.
- Il `T[]` restituito da `.ToArray()` utilizza meno memoria rispetto a `List<T>` restituito da `.ToList()` , quindi se il risultato verrà memorizzato per un lungo periodo, preferisci `.ToArray()` . Calling `List<T>.TrimExcess()` renderebbe la differenza di memoria strettamente accademica, al costo di eliminare il vantaggio relativo della velocità di `.ToList()` .

Examples

Seleziona (mappa)

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var names = persons.Select(p => p.Name);
Console.WriteLine(string.Join(",", names.ToArray()));

//Foo,Bar,Fizz,Buzz
```

Questo tipo di funzione viene solitamente chiamato `map` nei linguaggi di programmazione funzionale.

Dove (filtro)

Questo metodo restituisce un oggetto IEnumerable con tutti gli elementi che soddisfano l'espressione lambda

Esempio

```
var personNames = new[]
{
    "Foo", "Bar", "Fizz", "Buzz"
};

var namesStartingWithF = personNames.Where(p => p.StartsWith("F"));
Console.WriteLine(string.Join(",", namesStartingWithF));
```

Produzione:

Foo, Fizz

Visualizza la demo

Ordinato da

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByName = persons.OrderBy(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Id).ToArray()));

//2,4,3,1
```

OrderByDescending

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByNameDescending = persons.OrderByDescending(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByNameDescending.Select(p =>
p.Id).ToArray()));

//1,3,4,2
```

contiene

```
var numbers = new[] {1,2,3,4,5};
Console.WriteLine(numbers.Contains(3)); //True
Console.WriteLine(numbers.Contains(34)); //False
```

tranne

```
var numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbersBetweenSixAndFourteen = new[] { 6, 8, 10, 12 };

var result = numbers.Except(evenNumbersBetweenSixAndFourteen);

Console.WriteLine(string.Join(",", result));

//1, 2, 3, 4, 5, 7, 9
```

intersecare

```
var numbers1to10 = new[] {1,2,3,4,5,6,7,8,9,10};
var numbers5to15 = new[] {5,6,7,8,9,10,11,12,13,14,15};

var numbers5to10 = numbers1to10.Intersect(numbers5to15);

Console.WriteLine(string.Join(",", numbers5to10));

//5,6,7,8,9,10
```

concat

```
var numbers1to5 = new[] {1, 2, 3, 4, 5};
var numbers4to8 = new[] {4, 5, 6, 7, 8};

var numbers1to8 = numbers1to5.Concat(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,4,5,6,7,8
```

Si noti che i duplicati vengono mantenuti nel risultato. Se questo non è desiderabile, usa invece `Union` .

Primo (trova)

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.First();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.First(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2
```

Il seguente lancio di `InvalidOperationException` con il messaggio "Sequenza non contiene elementi corrispondenti":

```
var firstNegativeNumber = numbers.First(n => n < 0);
```

singolo

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.Single();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.Single(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1
```

Di seguito viene `InvalidOperationException` poiché è presente più di un elemento nella sequenza:

```
var theOnlyNumberInNumbers = numbers.Single();
var theOnlyNegativeNumber = numbers.Single(n => n < 0);
```

Scorso

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.Last();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.Last(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4
```

Il seguente lancio di `InvalidOperationException` :

```
var lastNegativeNumber = numbers.Last(n => n < 0);
```

LastOrDefault

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.LastOrDefault();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.LastOrDefault(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4

var lastNegativeNumber = numbers.LastOrDefault(n => n < 0);
Console.WriteLine(lastNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var lastWord = words.LastOrDefault();
Console.WriteLine(lastWord); // five
```

```
var lastLongWord = words.LastOrDefault(w => w.Length > 4);
Console.WriteLine(lastLongWord); // three

var lastMissingWord = words.LastOrDefault(w => w.Length > 5);
Console.WriteLine(lastMissingWord); // null
```

SingleOrDefault

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.SingleOrDefault();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.SingleOrDefault(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1

var theOnlyNegativeNumber = numbers.SingleOrDefault(n => n < 0);
Console.WriteLine(theOnlyNegativeNumber);  //0
```

Il seguente lancio di `InvalidOperationException` :

```
var theOnlyNumberInNumbers = numbers.SingleOrDefault();
```

FirstOrDefault

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.FirstOrDefault();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.FirstOrDefault(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2

var firstNegativeNumber = numbers.FirstOrDefault(n => n < 0);
Console.WriteLine(firstNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var firstWord = words.FirstOrDefault();
Console.WriteLine(firstWord); // one

var firstLongWord = words.FirstOrDefault(w => w.Length > 3);
Console.WriteLine(firstLongWord); // three

var firstMissingWord = words.FirstOrDefault(w => w.Length > 5);
Console.WriteLine(firstMissingWord); // null
```

Qualunque

Restituisce `true` se la raccolta ha elementi che soddisfano la condizione nell'espressione lambda:

```
var numbers = new[] {1,2,3,4,5};
```

```
var isNotEmpty = numbers.Any();
Console.WriteLine(isNotEmpty); //True

var anyNumberIsOne = numbers.Any(n => n == 1);
Console.WriteLine(anyNumberIsOne); //True

var anyNumberIsSix = numbers.Any(n => n == 6);
Console.WriteLine(anyNumberIsSix); //False

var anyNumberIsOdd = numbers.Any(n => (n & 1) == 1);
Console.WriteLine(anyNumberIsOdd); //True

var anyNumberIsNegative = numbers.Any(n => n < 0);
Console.WriteLine(anyNumberIsNegative); //False
```

Tutti

```
var numbers = new[] {1,2,3,4,5};

var allNumbersAreOdd = numbers.All(n => (n & 1) == 1);
Console.WriteLine(allNumbersAreOdd); //False

var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

Si noti che `All` funzioni del metodo controllano che il primo elemento venga valutato come `false` base al predicato. Pertanto, il metodo restituirà `true` per *qualsiasi* predicato nel caso in cui il set sia vuoto:

```
var numbers = new int[0];
var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

SelectMany (mappa piana)

`Enumerable.Select` restituisce un elemento di output per ogni elemento di input. Mentre `Enumerable.SelectMany` produce un numero variabile di elementi di output per ciascun elemento di input. Ciò significa che la sequenza di output può contenere più o meno elementi rispetto alla sequenza di input.

`Lambda expressions` passate a `Enumerable.Select` devono restituire un singolo elemento. Le espressioni lambda passate a `Enumerable.SelectMany` devono produrre una sequenza figlio. Questa sequenza figlio può contenere un numero variabile di elementi per ciascun elemento nella sequenza di input.

Esempio

```
class Invoice
{
    public int Id { get; set; }
}
```

```
class Customer
{
    public Invoice[] Invoices {get;set;}
}

var customers = new[] {
    new Customer {
        Invoices = new[] {
            new Invoice {Id=1},
            new Invoice {Id=2},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=3},
            new Invoice {Id=4},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=5},
            new Invoice {Id=6},
        }
    }
};

var allInvoicesFromAllCustomers = customers.SelectMany(c => c.Invoices);

Console.WriteLine(
    string.Join(",", allInvoicesFromAllCustomers.Select(i => i.Id).ToArray()));
```

Produzione:

1,2,3,4,5,6

Visualizza la demo

`Enumerable.SelectMany` può essere ottenuta anche con una query sintassi basata utilizzando due consecutivi `from` clausole:

```
var allInvoicesFromAllCustomers
    = from customer in customers
      from invoice in customer.Invoices
      select invoice;
```

Somma

```
var numbers = new[] {1,2,3,4};

var sumOfAllNumbers = numbers.Sum();
Console.WriteLine(sumOfAllNumbers); //10

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};
```

```
var totalPopulation = cities.Sum(c => c.Population);
Console.WriteLine(totalPopulation); //7500
```

Salta

Salta enumera i primi N elementi senza restituirli. Una volta raggiunto il numero di articolo N + 1, Skip inizia a restituire ogni elemento elencato:

```
var numbers = new[] {1,2,3,4,5};

var allNumbersExceptFirstTwo = numbers.Skip(2);
Console.WriteLine(string.Join(",", allNumbersExceptFirstTwo.ToArray()));

//3,4,5
```

Prendere

Questo metodo prende i primi `n` elementi da una enumerabile.

```
var numbers = new[] {1,2,3,4,5};

var threeFirstNumbers = numbers.Take(3);
Console.WriteLine(string.Join(",", threeFirstNumbers.ToArray()));

//1,2,3
```

SequenceEqual

```
var numbers = new[] {1,2,3,4,5};
var sameNumbers = new[] {1,2,3,4,5};
var sameNumbersInDifferentOrder = new[] {5,1,4,2,3};

var equalIfSameOrder = numbers.SequenceEqual(sameNumbers);
Console.WriteLine(equalIfSameOrder); //True

var equalIfDifferentOrder = numbers.SequenceEqual(sameNumbersInDifferentOrder);
Console.WriteLine(equalIfDifferentOrder); //False
```

Inverso

```
var numbers = new[] {1,2,3,4,5};
var reversed = numbers.Reverse();

Console.WriteLine(string.Join(",", reversed.ToArray()));

//5,4,3,2,1
```

OfType

```
var mixed = new object[] {1,"Foo",2,"Bar",3,"Fizz",4,"Buzz"};
```

```
var numbers = mixed.OfType<int>();

Console.WriteLine(string.Join(",", numbers.ToArray()));

//1,2,3,4
```

Max

```
var numbers = new[] {1,2,3,4};

var maxNumber = numbers.Max();
Console.WriteLine(maxNumber); //4

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var maxPopulation = cities.Max(c => c.Population);
Console.WriteLine(maxPopulation); //4000
```

min

```
var numbers = new[] {1,2,3,4};

var minNumber = numbers.Min();
Console.WriteLine(minNumber); //1

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var minPopulation = cities.Min(c => c.Population);
Console.WriteLine(minPopulation); //1000
```

Media

```
var numbers = new[] {1,2,3,4};

var averageNumber = numbers.Average();
Console.WriteLine(averageNumber);
// 2,5
```

Questo metodo calcola la media di enumerable di numeri.

```
var cities = new[] {
    new {Population = 1000},
    new {Population = 2000},
    new {Population = 4000}
};
```

```
var averagePopulation = cities.Average(c => c.Population);
Console.WriteLine(averagePopulation);
// 2333,33
```

Questo metodo calcola la media di enumerable utilizzando la funzione delegata.

Cerniera lampo

.NET 4.0

```
var tens = new[] {10,20,30,40,50};
var units = new[] {1,2,3,4,5};

var sums = tens.Zip(units, (first, second) => first + second);

Console.WriteLine(string.Join(",", sums));

//11,22,33,44,55
```

distinto

```
var numbers = new[] {1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
var distinctNumbers = numbers.Distinct();

Console.WriteLine(string.Join(",", distinctNumbers));

//1,2,3,4,5
```

Raggruppa per

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.GroupBy(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Raggruppa le fatture per paese, generando un nuovo oggetto con il numero di record, totale pagato e retribuzione media

```
var a = db.Invoices.GroupBy(i => i.Country)
```

```
          .Select(g => new { Country = g.Key,
                             Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

Se vogliamo solo i totali, nessun gruppo

```
var a = db.Invoices.GroupBy(i => 1)
          .Select(g => new { Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

Se abbiamo bisogno di più conteggi

```
var a = db.Invoices.GroupBy(g => 1)
          .Select(g => new { High = g.Count(i => i.Paid >= 1000),
                             Low = g.Count(i => i.Paid < 1000),
                             Sum = g.Sum(i => i.Paid) });
```

ToDictionary

Restituisce un nuovo dizionario dalla sorgente `IEnumerable` utilizzando la funzione keySelector fornita per determinare le chiavi. Lancia una `ArgumentException` se keySelector non è iniettivo (restituisce un valore univoco per ogni membro della collezione di origine.) Esistono sovraccarichi che consentono di specificare il valore da memorizzare e la chiave.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
};
```

Specificando solo una funzione selettore di chiave creerai un `Dictionary<TKey,TVal>` con `TKey` il tipo di ritorno del selettore di chiave, `TVal` il tipo di oggetto originale e l'oggetto originale come valore memorizzato.

```
var personsById = persons.ToDictionary(p => p.Id);
// personsById is a Dictionary<int,object>

Console.WriteLine(personsById[1].Name); //Fizz
Console.WriteLine(personsById[2].Name); //Buzz
```

Specificando anche una funzione selettore valore creerai un `Dictionary<TKey,TVal>` con `TKey` ancora il tipo di ritorno del selettore di chiave, ma `TVal` ora il tipo restituito della funzione selettore valore e il valore restituito come valore memorizzato.

```
var namesById = persons.ToDictionary(p => p.Id, p => p.Name);
//namesById is a Dictionary<int,string>

Console.WriteLine(namesById[3]); //Foo
```

```
Console.WriteLine(namesById[4]); //Bar
```

Come detto sopra, le chiavi restituite dal selettore di chiave devono essere uniche. Quanto segue genererà un'eccezione.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
    new { Name="Oops", Id=4}
};

var willThrowException = persons.ToDictionary(p => p.Id)
```

Se non è possibile fornire una chiave univoca per la raccolta di origine, prendere in considerazione l'utilizzo di ToLookup. In superficie, ToLookup si comporta in modo simile a ToDictionary, tuttavia, nella ricerca risultante ogni chiave è abbinata a un insieme di valori con chiavi corrispondenti.

Unione

```
var numbers1to5 = new[] {1,2,3,4,5};
var numbers4to8 = new[] {4,5,6,7,8};

var numbers1to8 = numbers1to5.Union(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,6,7,8
```

Si noti che i duplicati vengono rimossi dal risultato. Se questo non è auspicabile, utilizzare invece `Concat` .

ToArray

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersArray = someNumbers.ToArray();

Console.WriteLine(someNumbersArray.GetType().Name);
//Int32[]
```

Elencare

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);
```

```
Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersList = someNumbers.ToList();

Console.WriteLine(
    someNumbersList.GetType().Name + " - " +
    someNumbersList.GetType().GetGenericArguments()[0].Name);
//List`1 - Int32
```

Contare

```
IEnumerable<int> numbers = new[] {1,2,3,4,5,6,7,8,9,10};

var numbersCount = numbers.Count();
Console.WriteLine(numbersCount); //10

var evenNumbersCount = numbers.Count(n => (n & 1) == 0);
Console.WriteLine(evenNumbersCount); //5
```

ElementAt

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAt(2);
Console.WriteLine(thirdName); //Fizz

//The following throws ArgumentOutOfRangeException

var minusOnethName = names.ElementAt(-1);
var fifthName = names.ElementAt(4);
```

ElementAtOrDefault

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAtOrDefault(2);
Console.WriteLine(thirdName); //Fizz

var minusOnethName = names.ElementAtOrDefault(-1);
Console.WriteLine(minusOnethName); //null

var fifthName = names.ElementAtOrDefault(4);
Console.WriteLine(fifthName); //null
```

SkipWhile

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var oddNumbers = numbers.SkipWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", oddNumbers.ToArray()));

//1,3,5,7
```

TakeWhile

```
var numbers = new[] {2,4,6,1,3,5,7,8};

var evenNumbers = numbers.TakeWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", evenNumbers.ToArray()));

//2,4,6
```

DefaultIfEmpty

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var numbersOrDefault = numbers.DefaultIfEmpty();
Console.WriteLine(numbers.SequenceEqual(numbersOrDefault)); //True

var noNumbers = new int[0];

var noNumbersOrDefault = noNumbers.DefaultIfEmpty();
Console.WriteLine(noNumbersOrDefault.Count()); //1
Console.WriteLine(noNumbersOrDefault.Single()); //0

var noNumbersOrExplicitDefault = noNumbers.DefaultIfEmpty(34);
Console.WriteLine(noNumbersOrExplicitDefault.Count()); //1
Console.WriteLine(noNumbersOrExplicitDefault.Single()); //34
```

Aggregato (piega)

Generazione di un nuovo oggetto in ogni fase:

```
var elements = new[] {1,2,3,4,5};

var commaSeparatedElements = elements.Aggregate(
    seed: "",
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements);  //1,2,3,4,5,
```

Utilizzando lo stesso oggetto in tutti i passaggi:

```
var commaSeparatedElements2 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"));

Console.WriteLine(commaSeparatedElements2.ToString());  //1,2,3,4,5,
```

Utilizzando un selettore di risultati:

```
var commaSeparatedElements3 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"),
    resultSelector: (seed) => seed.ToString());
```

```
Console.WriteLine(commaSeparatedElements3);  //1,2,3,4,5,
```

Se un seme viene omesso, il primo elemento diventa il seme:

```
var seedAndElements = elements.Select(n=>n.ToString());
var commaSeparatedElements4 = seedAndElements.Aggregate(
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements4);  //12,3,4,5,
```

ToLookup

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.ToLookup(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Aderire

```
class Developer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
        Id = 2,
        Name = "Barfizz"
    }
};
```

```
var projects = new[] {
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var denormalized = developers.Join(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, proj) => new {
            ProjectName = proj.Name,
            DeveloperName = dev.Name});

foreach(var item in denormalized)
{
    Console.WriteLine("{0} by {1}", item.ProjectName, item.DeveloperName);
}

//Hello World 3D by Foobuzz
//Super Fizzbuzz Maker by Foobuzz
//Citizen Kane - The action game by Barfizz
//Pro Pong 2016 by Barfizz
```

GroupJoin

```
class Developer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
```

```
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var grouped = developers.GroupJoin(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, projs) => new {
            DeveloperName = dev.Name,
            ProjectNames = projs.Select(p => p.Name).ToArray()});

foreach(var item in grouped)
{
    Console.WriteLine(
        "{0}'s projects: {1}",
        item.DeveloperName,
        string.Join(", ", item.ProjectNames));
}

//Foobuzz's projects: Hello World 3D, Super Fizzbuzz Maker
//Barfizz's projects: Citizen Kane - The action game, Pro Pong 2016
```

lanciare

`Cast` è diverso dagli altri metodi di `Enumerable` in quanto è un metodo di estensione per `IEnumerable` , non per `IEnumerable<T>` . Quindi può essere usato per convertire istanze del precedente in istanze del successivo.

Questo non viene compilato poiché `ArrayList` non implementa `IEnumerable<T>` :

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.First());
```

Funziona come previsto:

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.Cast<int>().First()); //1
```

`Cast` **non** esegue cast di conversione. Il seguente compila, ma lancia `InvalidCastException` in fase di runtime:

```
var numbers = new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Cast<decimal>().ToArray();
```

Il modo corretto per eseguire un cast di conversione in una raccolta è il seguente:

```
var numbers= new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Select(n => (decimal)n).ToArray();
```

Vuoto

Per creare un IEnumerable vuoto di int:

```
IEnumerable<int> emptyList = Enumerable.Empty<int>();
```

Questo IEnumerable vuoto viene memorizzato nella cache per ogni tipo T, in modo che:

```
Enumerable.Empty<decimal>() == Enumerable.Empty<decimal>(); // This is True
Enumerable.Empty<int>() == Enumerable.Empty<decimal>();     // This is False
```

ThenBy

`ThenBy` può essere utilizzato solo dopo una clausola `OrderBy` che consente di ordinare utilizzando più criteri

```
var persons = new[]
{
    new {Id = 1, Name = "Foo", Order = 1},
    new {Id = 1, Name = "FooTwo", Order = 2},
    new {Id = 2, Name = "Bar", Order = 2},
    new {Id = 2, Name = "BarTwo", Order = 1},
    new {Id = 3, Name = "Fizz", Order = 2},
    new {Id = 3, Name = "FizzTwo", Order = 1},
};

var personsSortedByName = persons.OrderBy(p => p.Id).ThenBy(p => p.Order);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Name)));
//This will display :
//Foo,FooTwo,BarTwo,Bar,FizzTwo,Fizz
```

Gamma

I due parametri di `Range` sono il *primo* numero e il *numero* di elementi da produrre (non l'ultimo numero).

```
// prints 1,2,3,4,5,6,7,8,9,10
Console.WriteLine(string.Join(",", Enumerable.Range(1, 10)));

// prints 10,11,12,13,14
Console.WriteLine(string.Join(",", Enumerable.Range(10, 5)));
```

Left Outer Join

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void Main(string[] args)
{
    var magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    var terry = new Person { FirstName = "Terry", LastName = "Adams" };

    var barley = new Pet { Name = "Barley", Owner = terry };

    var people = new[] { magnus, terry };
    var pets = new[] { barley };

    var query =
        from person in people
        join pet in pets on person equals pet.Owner into gj
        from subpet in gj.DefaultIfEmpty()
        select new
        {
            person.FirstName,
            PetName = subpet?.Name ?? "-" // Use - if he has no pet
        };

    foreach (var p in query)
        Console.WriteLine($"{p.FirstName}: {p.PetName}");
}
```

Ripetere

`Enumerable.Repeat` genera una sequenza di un valore ripetuto. In questo esempio genera "Ciao" 4 volte.

```
var repeats = Enumerable.Repeat("Hello", 4);

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

/* output:
```

```
    Hello
    Hello
    Hello
    Hello
*/
```

Leggi LINQ online: https://riptutorial.com/it/dot-net/topic/34/linq

Capitolo 30: Managed Extensibility Framework

Osservazioni

Uno dei grandi vantaggi del MEF rispetto ad altre tecnologie che supportano il pattern di inversione di controllo è che supporta la risoluzione di dipendenze non note in fase di progettazione, senza la necessità di una configurazione (se non nulla).

Tutti gli esempi richiedono un riferimento all'assembly System.ComponentModel.Composition.

Inoltre, tutti gli esempi (di base) li utilizzano come oggetti di business di esempio:

```
using System.Collections.ObjectModel;

namespace Demo
{
    public sealed class User
    {
        public User(int id, string name)
        {
            this.Id = id;
            this.Name = name;
        }

        public int Id { get; }
        public string Name { get; }
        public override string ToString() => $"User[Id: {this.Id}, Name={this.Name}]";
    }

    public interface IUserProvider
    {
        ReadOnlyCollection<User> GetAllUsers();
    }
}
```

Examples

Esportare un tipo (base)

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel.Composition;

namespace Demo
{
    [Export(typeof(IUserProvider))]
    public sealed class UserProvider : IUserProvider
    {
        public ReadOnlyCollection<User> GetAllUsers()
        {
```

```
            return new List<User>
            {
                new User(0, "admin"),
                new User(1, "Dennis"),
                new User(2, "Samantha"),
            }.AsReadOnly();
        }
    }
}
```

Questo potrebbe essere definito praticamente ovunque; tutto ciò che conta è che l'applicazione sappia dove cercarlo (tramite ComposablePartCatalogs che crea).

Importazione (base)

```
using System;
using System.ComponentModel.Composition;

namespace Demo
{
    public sealed class UserWriter
    {
        [Import(typeof(IUserProvider))]
        private IUserProvider userProvider;

        public void PrintAllUsers()
        {
            foreach (User user in this.userProvider.GetAllUsers())
            {
                Console.WriteLine(user);
            }
        }
    }
}
```

Questo è un tipo che ha una dipendenza da un `IUserProvider` , che può essere definito ovunque. Come nell'esempio precedente, tutto ciò che conta è che l'applicazione sappia dove cercare l'esportazione corrispondente (tramite ComposablePartCatalogs che crea).

Connessione (base)

Vedi gli altri esempi (di base) sopra.

```
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;

namespace Demo
{
    public static class Program
    {
        public static void Main()
        {
            using (var catalog = new ApplicationCatalog())
            using (var exportProvider = new CatalogExportProvider(catalog))
            using (var container = new CompositionContainer(exportProvider))
```

```
            {
                exportProvider.SourceProvider = container;

                UserWriter writer = new UserWriter();

                // at this point, writer's userProvider field is null
                container.ComposeParts(writer);

                // now, it should be non-null (or an exception will be thrown).
                writer.PrintAllUsers();
            }
        }
    }
}
```

Finché qualcosa nel percorso di ricerca dell'assembly dell'applicazione ha `[Export(typeof(IUserProvider))]` , l'importazione corrispondente di `UserWriter` sarà soddisfatta e gli utenti verranno stampati.

Altri tipi di cataloghi (ad esempio `DirectoryCatalog`) possono essere utilizzati al posto di (o in aggiunta a) `ApplicationCatalog` , per cercare in altri luoghi le esportazioni che soddisfano le importazioni.

Leggi Managed Extensibility Framework online: https://riptutorial.com/it/dot-net/topic/62/managed-extensibility-framework

Capitolo 31: Moduli VB

Examples

Ciao mondo in moduli VB.NET

Per mostrare una finestra di messaggio quando il modulo è stato mostrato:

```
Public Class Form1
    Private Sub Form1_Shown(sender As Object, e As EventArgs) Handles MyBase.Shown
        MessageBox.Show("Hello, World!")
    End Sub
End Class
To show a message box before the form has been shown:

Public Class Form1
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        MessageBox.Show("Hello, World!")
    End Sub
End Class
```

Load () verrà chiamato prima e solo una volta, quando il modulo viene caricato per la prima volta. Show () verrà chiamato ogni volta che l'utente avvia il modulo. Activate () verrà chiamato ogni volta che l'utente attiva il modulo.

Load () verrà eseguito prima che venga chiamato Show (), ma attenzione: chiamare msgBox () in show può causare che msgBox () venga eseguito prima che Load () abbia finito. **In genere è una cattiva idea dipendere dall'ordine degli eventi tra Load (), Show () e simili.**

Per principianti

Alcune cose che tutti i principianti dovrebbero sapere / fanno che li aiuteranno ad avere un buon inizio con VB. Net:

Imposta le seguenti opzioni:

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

End Class
```

Usa &, non + per la concatenazione di stringhe. Le stringhe dovrebbero essere studiate in dettaglio in quanto sono ampiamente utilizzate.

Dedica un po 'di tempo alla comprensione del valore e dei tipi di riferimento .

Non utilizzare mai Application.DoEvents . Prestare attenzione a 'Attenzione'. Quando raggiungi un punto in cui questo sembra qualcosa che devi usare, chiedi.

La documentazione è tua amica.

Timer delle forme

Il componente Windows.Forms.Timer può essere utilizzato per fornire all'utente informazioni che **non sono** critiche dal punto di vista del tempo. Crea un modulo con un pulsante, un'etichetta e un componente Timer.

Ad esempio potrebbe essere usato per mostrare periodicamente all'utente l'ora del giorno.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 60 * 1000 'one minute intervals
        'start timer
        Timer1.Start()
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub
End Class
```

Ma questo timer non è adatto per i tempi. Un esempio potrebbe usarlo per un conto alla rovescia. In questo esempio simuleremo un conto alla rovescia per tre minuti. Questo potrebbe essere uno degli esempi più noiosamente importanti qui.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        ctSecs = 0 'clear count
        Timer1.Interval = 1000 'one second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
    End Sub
```

```
    Dim stpw As New Stopwatch
    Dim ctSecs As Integer

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        ctSecs += 1
        If ctSecs = 180 Then 'about 2.5 seconds off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'Is it near 180?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

Dopo aver fatto clic sul pulsante 1, passano circa tre minuti e label1 mostra i risultati. Label1 mostra 180? Probabilmente no. Sulla mia macchina ha mostrato 182,5!

Il motivo della discrepanza è nella documentazione: "Il componente Timer di Windows Form è a thread singolo ed è limitato a una precisione di 55 millisecondi." Questo è il motivo per cui non dovrebbe essere usato per i tempi.

Utilizzando il cronometro e il cronometro in modo leggermente diverso possiamo ottenere risultati migliori.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 100 'one tenth of a second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
    End Sub

    Dim stpw As New Stopwatch
    Dim threeMinutes As TimeSpan = TimeSpan.FromMinutes(3)

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        If stpw.Elapsed >= threeMinutes Then '0.1 off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'how close?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

Ci sono altri timer che possono essere utilizzati secondo necessità. Questa ricerca dovrebbe aiutare in tal senso.

Leggi Moduli VB online: https://riptutorial.com/it/dot-net/topic/2197/moduli-vb

Capitolo 32: Networking

Osservazioni

Vedi anche: Client HTTP

Examples

Chat TCP di base (TcpListener, TcpClient, NetworkStream)

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpChat
{
    static void Main(string[] args)
    {
        if(args.Length == 0)
        {
            Console.WriteLine("Basic TCP chat");
            Console.WriteLine();
            Console.WriteLine("Usage:");
            Console.WriteLine("tcpchat server <port>");
            Console.WriteLine("tcpchat client <url> <port>");
            return;
        }

        try
        {
            Run(args);
        }
        catch(IOException)
        {
            Console.WriteLine("--- Connection lost");
        }
        catch(SocketException ex)
        {
            Console.WriteLine("--- Can't connect: " + ex.Message);
        }
    }

    static void Run(string[] args)
    {
        TcpClient client;
        NetworkStream stream;
        byte[] buffer = new byte[256];
        var encoding = Encoding.ASCII;

        if(args[0].StartsWith("s", StringComparison.InvariantCultureIgnoreCase))
        {
            var port = int.Parse(args[1]);
            var listener = new TcpListener(IPAddress.Any, port);
```

```
            listener.Start();
            Console.WriteLine("--- Waiting for a connection...");
            client = listener.AcceptTcpClient();
        }
        else
        {
            var hostName = args[1];
            var port = int.Parse(args[2]);
            client = new TcpClient();
            client.Connect(hostName, port);
        }

        stream = client.GetStream();
        Console.WriteLine("--- Connected. Start typing! (exit with Ctrl-C)");

        while(true)
        {
            if(Console.KeyAvailable)
            {
                var lineToSend = Console.ReadLine();
                var bytesToSend = encoding.GetBytes(lineToSend + "\r\n");
                stream.Write(bytesToSend, 0, bytesToSend.Length);
                stream.Flush();
            }

            if (stream.DataAvailable)
            {
                var receivedBytesCount = stream.Read(buffer, 0, buffer.Length);
                var receivedString = encoding.GetString(buffer, 0, receivedBytesCount);
                Console.Write(receivedString);
            }
        }
    }
}
```

Client SNTP di base (UdpClient)

Vedi RFC 2030 per dettagli sul protocollo SNTP.

```
using System;
using System.Globalization;
using System.Linq;
using System.Net;
using System.Net.Sockets;

class SntpClient
{
    const int SntpPort = 123;
    static DateTime BaseDate = new DateTime(1900, 1, 1);

    static void Main(string[] args)
    {
        if(args.Length == 0) {
            Console.WriteLine("Simple SNTP client");
            Console.WriteLine();
            Console.WriteLine("Usage: sntpclient <sntp server url> [<local timezone>]");
            Console.WriteLine();
            Console.WriteLine("<local timezone>: a number between -12 and 12 as hours from
UTC");
```

```
            Console.WriteLine("(append .5 for an extra half an hour)");
            return;
        }

        double localTimeZoneInHours = 0;
        if(args.Length > 1)
            localTimeZoneInHours = double.Parse(args[1], CultureInfo.InvariantCulture);

        var udpClient = new UdpClient();
        udpClient.Client.ReceiveTimeout = 5000;

        var sntpRequest = new byte[48];
        sntpRequest[0] = 0x23; //LI=0 (no warning), VN=4, Mode=3 (client)

        udpClient.Send(
            dgram: sntpRequest,
            bytes: sntpRequest.Length,
            hostname: args[0],
            port: SntpPort);

        byte[] sntpResponse;
        try
        {
            IPEndPoint remoteEndpoint = null;
            sntpResponse = udpClient.Receive(ref remoteEndpoint);
        }
        catch(SocketException)
        {
            Console.WriteLine("*** No response received from the server");
            return;
        }

        uint numberOfSeconds;
        if(BitConverter.IsLittleEndian)
            numberOfSeconds = BitConverter.ToUInt32(
                sntpResponse.Skip(40).Take(4).Reverse().ToArray()
                ,0);
        else
            numberOfSeconds = BitConverter.ToUInt32(sntpResponse, 40);

        var date = BaseDate.AddSeconds(numberOfSeconds).AddHours(localTimeZoneInHours);

        Console.WriteLine(
            $"Current date in server: {date:yyyy-MM-dd HH:mm:ss}
UTC{localTimeZoneInHours:+0.#;-0.#;.}");
    }
}
```

Leggi Networking online: https://riptutorial.com/it/dot-net/topic/35/networking

Capitolo 33: Panoramiche API Task Parallel Library (TPL)

Osservazioni

La libreria Task parallela è costituita da tipi pubblici e API che semplificano notevolmente il processo di aggiunta del parallelismo e della concorrenza a un'applicazione. .Netto. TPL è stato introdotto in .Net 4 ed è il metodo consigliato per scrivere codice multi thread e parallelo.

TPL si occupa della pianificazione del lavoro, dell'affinità dei thread, del supporto per la cancellazione, della gestione dello stato e del bilanciamento del carico in modo che il programmatore possa concentrarsi sulla risoluzione dei problemi piuttosto che dedicarsi a dettagli comuni di basso livello.

Examples

Eseguire il lavoro in risposta a un clic del pulsante e aggiornare l'interfaccia utente

Questo esempio dimostra come è possibile rispondere a un clic del pulsante eseguendo un lavoro su un thread di lavoro e quindi aggiornare l'interfaccia utente per indicare il completamento

```
void MyButton_OnClick(object sender, EventArgs args)
{
    Task.Run(() => // Schedule work using the thread pool
        {
            System.Threading.Thread.Sleep(5000); // Sleep for 5 seconds to simulate work.
        })
    .ContinueWith(p => // this continuation contains the 'update' code to run on the UI thread
    {
        this.TextBlock_ResultText.Text = "The work completed at " + DateTime.Now.ToString()
    },
    TaskScheduler.FromCurrentSynchronizationContext()); // make sure the update is run on the
UI thread.

}
```

Leggi Panoramiche API Task Parallel Library (TPL) online: https://riptutorial.com/it/dot-net/topic/5164/panoramiche-api-task-parallel-library--tpl-

Capitolo 34: Per ciascuno

Osservazioni

Usalo affatto?

Si potrebbe obiettare che l'intenzione del framework .NET è che le query non abbiano effetti collaterali e che il metodo `ForEach` sia per definizione causa di un effetto collaterale. Potresti trovare il tuo codice più manutenibile e più facile da testare se usi una `foreach` normale.

Examples

Chiamare un metodo su un oggetto in un elenco

```
public class Customer {
   public void SendEmail()
   {
       // Sending email code here
   }
}

List<Customer> customers = new List<Customer>();

customers.Add(new Customer());
customers.Add(new Customer());

customers.ForEach(c => c.SendEmail());
```

Metodo di estensione per IEnumerable

`ForEach()` è definito nella classe `List<T>` , ma non su `IQueryable<T>` o `IEnumerable<T>` . Hai due scelte in questi casi:

Per prima cosa

L'enumerazione (o la query) sarà valutata, copiando i risultati in un nuovo elenco o chiamando il database. Il metodo viene quindi chiamato su ciascun elemento.

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ToList().ForEach(c => c.SendEmail());
```

Questo metodo ha un overhead di utilizzo della memoria evidente, in quanto viene creato un elenco intermedio.

Metodo di estensione

Scrivi un metodo di estensione:

```
public static void ForEach<T>(this IEnumerable<T> enumeration, Action<T> action)
{
    foreach(T item in enumeration)
    {
        action(item);
    }
}
```

Uso:

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ForEach(c => c.SendEmail());
```

Attenzione: i metodi LINQ del Framework sono stati progettati con l'intento di essere *puri* , il che significa che non producono effetti collaterali. L'unico scopo del metodo `ForEach` è produrre effetti collaterali e devia dagli altri metodi sotto questo aspetto. Si può prendere in considerazione solo l'uso di un ciclo `foreach` normale invece.

Leggi Per ciascuno online: https://riptutorial.com/it/dot-net/topic/2225/per-ciascuno

Capitolo 35: Platform Invoke

Sintassi

- [DllImport ("Example.dll")] statico extern void SetText (string inString);
- [DllImport ("Example.dll")] statico extern void GetText (StringBuilder outString);
- [MarshalAs (UnmanagedType.ByValTStr, SizeConst = 32)] stringa di testo;
- [MarshalAs (UnmanagedType.ByValArray, SizeConst = 128)] byte [] byteArr;
- [StructLayout (LayoutKind.Sequential)] public struct PERSON {...}
- [StructLayout (LayoutKind.Explicit)] struttura pubblica MarshaledUnion {[FieldOffset (0)] ...}

Examples

Chiamando una funzione dll Win32

```
using System.Runtime.InteropServices;

class PInvokeExample
{
    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    public static extern uint MessageBox(IntPtr hWnd, String text, String caption, int
options);

    public static void test()
    {
        MessageBox(IntPtr.Zero, "Hello!", "Message", 0);
    }
}
```

Dichiarare una funzione come `static extern` stting `DllImportAttribute` con la relativa proprietà `Value` impostata sul nome .dll. Non dimenticare di utilizzare lo spazio dei nomi `System.Runtime.InteropServices` . Quindi chiamalo come un metodo statico regolare.

Il Platform Invocation Services si occuperà di caricare il file .dll e di trovare la tariffa desiderata. Il P / Invoke nella maggior parte dei casi semplici esegue anche il marshalling dei parametri e restituisce il valore da e verso la DLL (ovvero conversione da tipi di dati .NET a Win32 e viceversa).

Utilizzando l'API di Windows

Usa pinvoke.net .

Prima di dichiarare una funzione API `extern` Windows nel tuo codice, prova a cercarla su pinvoke.net . Molto probabilmente hanno già una dichiarazione adeguata con tutti i tipi di supporto e buoni esempi.

Matrici di marshalling

Matrici di tipo semplice

```
[DllImport("Example.dll")]
static extern void SetArray(
    [MarshalAs(UnmanagedType.LPArray, SizeConst = 128)]
    byte[] data);
```

Matrici di stringa

```
[DllImport("Example.dll")]
static extern void SetStrArray(string[] textLines);
```

Strutture di marshalling

Struttura semplice

Firma C ++:

```
typedef struct _PERSON
{
    int age;
    char name[32];
} PERSON, *LP_PERSON;

void GetSpouse(PERSON person, LP_PERSON spouse);
```

Definizione C #

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct PERSON
{
    public int age;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string name;
}

[DllImport("family.dll", CharSet = CharSet.Auto)]
public static extern bool GetSpouse(PERSON person, ref PERSON spouse);
```

Strutturare con campi array di dimensioni sconosciute. Passando dentro

Firma C ++

```
typedef struct
{
    int length;
    int *data;
} VECTOR;

void SetVector(VECTOR &vector);
```

Passato da codice gestito a codice non gestito, questo

L'array di `data` deve essere definito come IntPtr e la memoria deve essere allocata esplicitamente con `Marshal.AllocHGlobal()` (e liberata con `Marshal.FreeHGlobal()`):

```
[StructLayout(LayoutKind.Sequential)]
public struct VECTOR : IDisposable
{
    int length;
    IntPtr dataBuf;

    public int[] data
    {
        set
        {
            FreeDataBuf();
            if (value != null && value.Length > 0)
            {
                dataBuf = Marshal.AllocHGlobal(value.Length * Marshal.SizeOf(value[0]));
                Marshal.Copy(value, 0, dataBuf, value.Length);
                length = value.Length;
            }
        }
    }
    void FreeDataBuf()
    {
        if (dataBuf != IntPtr.Zero)
        {
            Marshal.FreeHGlobal(dataBuf);
            dataBuf = IntPtr.Zero;
        }
    }
    public void Dispose()
    {
        FreeDataBuf();
    }
}

[DllImport("vectors.dll")]
public static extern void SetVector([In]ref VECTOR vector);
```

Strutturare con campi array di dimensioni sconosciute. ricevente

Firma C ++:

```
typedef struct
{
    char *name;
} USER;

bool GetCurrentUser(USER *user);
```

Quando tali dati vengono passati dal codice non gestito e la memoria viene allocata dalle funzioni non gestite, il chiamante gestito deve riceverlo in una variabile `IntPrt` e convertire il buffer in un array gestito. In caso di stringhe, esiste un metodo `Marshal.PtrToStringAnsi()` conveniente:

```
[StructLayout(LayoutKind.Sequential)]
public struct USER
{
```

```
    IntPtr nameBuffer;
    public string name { get { return Marshal.PtrToStringAnsi(nameBuffer); } }
}

[DllImport("users.dll")]
public static extern bool GetCurrentUser(out USER user);
```

I sindacati di marshalling

Solo campi di tipo valore

Dichiarazione C ++

```
typedef union
{
    char c;
    int i;
} CharOrInt;
```

Dichiarazione C #

```
[StructLayout(LayoutKind.Explicit)]
public struct CharOrInt
{
    [FieldOffset(0)]
    public byte c;
    [FieldOffset(0)]
    public int i;
}
```

Miscelazione del tipo di valore e dei campi di riferimento

La sovrapposizione di un valore di riferimento con un tipo di valore 1 non è consentita, pertanto non è possibile utilizzare semplicemente il ~~`FieldOffset(0) text; FieldOffset(0) i;`~~ non verrà compilato per

```
typedef union
{
    char text[128];
    int i;
} TextOrInt;
```

e in genere è necessario utilizzare il marshalling personalizzato. Tuttavia, in casi particolari come questa tecnica più semplice si può usare:

```
[StructLayout(LayoutKind.Sequential)]
public struct TextOrInt
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 128)]
    public byte[] text;
    public int i { get { return BitConverter.ToInt32(text, 0); } }
}
```

Leggi Platform Invoke online: https://riptutorial.com/it/dot-net/topic/1643/platform-invoke

Capitolo 36: Porte seriali

Examples

Operazione base

```
var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
serialPort.Open();
serialPort.WriteLine("Test data");
string response = serialPort.ReadLine();
Console.WriteLine(response);
serialPort.Close();
```

Elenca i nomi delle porte disponibili

```
string[] portNames = SerialPort.GetPortNames();
```

Lettura asincrona

```
void SetupAsyncRead(SerialPort serialPort)
{
    serialPort.DataReceived += (sender, e) => {
        byte[] buffer = new byte[4096];
        switch (e.EventType)
        {
            case SerialData.Chars:
                var port = (SerialPort)sender;
                int bytesToRead = port.BytesToRead;
                if (bytesToRead > buffer.Length)
                    Array.Resize(ref buffer, bytesToRead);
                int bytesRead = port.Read(buffer, 0, bytesToRead);
                // Process the read buffer here
                // ...
                break;
            case SerialData.Eof:
                // Terminate the service here
                // ...
                break;
        }
    };
```

Servizio echo testo sincrono

```
using System.IO.Ports;

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

```
            var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
            serialPort.Open();
            string message = "";
            while (message != "quit")
            {
                message = serialPort.ReadLine();
                serialPort.WriteLine(message);
            }
            serialPort.Close();
        }
    }
}
```

Ricevitore di messaggi asincroni

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Text;
using System.Threading;

namespace AsyncReceiver
{
    class Program
    {
        const byte STX = 0x02;
        const byte ETX = 0x03;
        const byte ACK = 0x06;
        const byte NAK = 0x15;
        static ManualResetEvent terminateService = new ManualResetEvent(false);
        static readonly object eventLock = new object();
        static List<byte> unprocessedBuffer = null;

        static void Main(string[] args)
        {
            try
            {
                var serialPort = new SerialPort("COM11", 9600, Parity.Even, 8, StopBits.One);
                serialPort.DataReceived += DataReceivedHandler;
                serialPort.ErrorReceived += ErrorReceivedHandler;
                serialPort.Open();
                terminateService.WaitOne();
                serialPort.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception occurred: {0}", e.Message);
            }
            Console.ReadKey();
        }

        static void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
        {
            lock (eventLock)
            {
                byte[] buffer = new byte[4096];
                switch (e.EventType)
                {
                    case SerialData.Chars:
```

```
                    var port = (SerialPort)sender;
                    int bytesToRead = port.BytesToRead;
                    if (bytesToRead > buffer.Length)
                        Array.Resize(ref buffer, bytesToRead);
                    int bytesRead = port.Read(buffer, 0, bytesToRead);
                    ProcessBuffer(buffer, bytesRead);
                    break;
                case SerialData.Eof:
                    terminateService.Set();
                    break;
            }
        }
    }
    static void ErrorReceivedHandler(object sender, SerialErrorReceivedEventArgs e)
    {
        lock (eventLock)
            if (e.EventType == SerialError.TXFull)
            {
                Console.WriteLine("Error: TXFull. Can't handle this!");
                terminateService.Set();
            }
            else
            {
                Console.WriteLine("Error: {0}. Resetting everything", e.EventType);
                var port = (SerialPort)sender;
                port.DiscardInBuffer();
                port.DiscardOutBuffer();
                unprocessedBuffer = null;
                port.Write(new byte[] { NAK }, 0, 1);
            }
    }

    static void ProcessBuffer(byte[] buffer, int length)
    {
        List<byte> message = unprocessedBuffer;
        for (int i = 0; i < length; i++)
            if (buffer[i] == ETX)
            {
                if (message != null)
                {
                    Console.WriteLine("MessageReceived: {0}",
                        Encoding.ASCII.GetString(message.ToArray()));
                    message = null;
                }
            }
            else if (buffer[i] == STX)
                message = null;
            else if (message != null)
                message.Add(buffer[i]);
        unprocessedBuffer = message;
    }
}
}
```

Questo programma attende i messaggi racchiusi tra byte `STX` ed `ETX` e restituisce il testo che si frappone tra loro. Tutto il resto viene scartato. In caso di overflow del buffer di scrittura, si interrompe. Su altri errori resetta i buffer di input e output e attende ulteriori messaggi.

Il codice illustra:

- Lettura della porta seriale asincrona (vedi Uso `SerialPort.DataReceived`).
- Elaborazione dell'errore della porta seriale (vedere Uso `SerialPort.ErrorReceived`).
- Implementazione del protocollo basato su messaggi non testuali.
- Lettura parziale dei messaggi
    - L'evento `SerialPort.DataReceived` potrebbe verificarsi prima dell'intero messaggio (fino a `ETX`). L'intero messaggio potrebbe anche non essere disponibile nel buffer di input (SerialPort.Read (..., ..., port.BytesToRead) legge solo una parte del messaggio). In questo caso inseriamo la parte ricevuta (`unprocessedBuffer`) e continuiamo ad aspettare ulteriori dati.
- Trattare con diversi messaggi in arrivo.
    - L'evento `SerialPort.DataReceived` può verificarsi solo dopo che diversi messaggi sono stati inviati dall'altra parte.

Leggi Porte seriali online: https://riptutorial.com/it/dot-net/topic/5366/porte-seriali

Capitolo 37: Raccolta dei rifiuti

introduzione

In. Netto, gli oggetti creati con new () sono allocati nell'heap gestito. Questi oggetti non sono mai esplicitamente finalizzati dal programma che li usa; invece, questo processo è controllato da .Net Garbage Collector.

Alcuni degli esempi riportati di seguito sono "casi di laboratorio" per mostrare il Garbage Collector al lavoro e alcuni dettagli significativi del suo comportamento, mentre altri si concentrano su come preparare le classi per una corretta gestione da parte di Garbage Collector.

Osservazioni

Il Garbage Collector ha lo scopo di ridurre il costo del programma in termini di memoria allocata, ma farlo ha un costo in termini di tempo di elaborazione. Per raggiungere un buon compromesso generale, ci sono alcune ottimizzazioni che dovrebbero essere prese in considerazione durante la programmazione con il Garbage Collector in mente:

- Se il metodo Collect () deve essere invocato in modo esplicito (che comunque non dovrebbe essere il caso), si consideri l'utilizzo della modalità "ottimizzata" che finalizza l'oggetto morto solo quando la memoria è effettivamente necessaria
- Invece di invocare il metodo Collect (), considerare l'utilizzo dei metodi AddMemoryPressure () e RemoveMemoryPressure (), che attivano una raccolta di memoria solo se effettivamente necessario
- Una raccolta di memoria non è garantita per finalizzare tutti gli oggetti morti; invece, il Garbage Collector gestisce 3 "generazioni", un oggetto che a volte "sopravvive" da una generazione al successivo
- Possono essere applicati diversi modelli di threading, in base a vari fattori tra cui la messa a punto dell'ottimizzazione, con conseguenti diversi gradi di interferenza tra il thread Garbage Collector e gli altri thread dell'applicazione

Examples

Un esempio di base della raccolta (garbage)

Data la seguente classe:

```
public class FinalizableObject
{
    public FinalizableObject()
    {
        Console.WriteLine("Instance initialized");
    }

    ~FinalizableObject()
```

```
    {
        Console.WriteLine("Instance finalized");
    }
}
```

Un programma che crea un'istanza, anche senza usarlo:

```
new FinalizableObject(); // Object instantiated, ready to be used
```

Produce il seguente risultato:

```
<namespace>.FinalizableObject initialized
```

Se non accade nient'altro, l'oggetto non è finalizzato fino alla fine del programma (che libera tutti gli oggetti sull'heap gestito, finalizzandoli nel processo).

È possibile forzare l'esecuzione di Garbage Collector in un determinato punto, come indicato di seguito:

```
new FinalizableObject(); // Object instantiated, ready to be used
GC.Collect();
```

Che produce il seguente risultato:

```
<namespace>.FinalizableObject initialized
<namespace>.FinalizableObject finalized
```

Questa volta, non appena è stato richiamato il Garbage Collector, l'oggetto non utilizzato (ovvero "morto") è stato finalizzato e liberato dall'heap gestito.

Oggetti in diretta e oggetti morti - le basi

Regola pratica: quando si verifica la garbage collection, gli "oggetti live" sono quelli ancora in uso, mentre gli "oggetti morti" sono quelli non più utilizzati (qualsiasi variabile o campo che li fa riferimento, se presente, è andato fuori campo prima che si verifichi la raccolta) .

Nell'esempio seguente (per praticità, FinalizableObject1 e FinalizableObject2 sono sottoclassi di FinalizableObject dell'esempio precedente e quindi ereditano il comportamento del messaggio di inizializzazione / finalizzazione):

```
var obj1 = new FinalizableObject1(); // Finalizable1 instance allocated here
var obj2 = new FinalizableObject2(); // Finalizable2 instance allocated here
obj1 = null; // No more references to the Finalizable1 instance
GC.Collect();
```

L'output sarà:

```
<namespace>.FinalizableObject1 initialized
<namespace>.FinalizableObject2 initialized
```

```
<namespace>.FinalizableObject1 finalized
```

Nel momento in cui viene richiamato il Garbage Collector, FinalizableObject1 è un oggetto morto e viene finalizzato, mentre FinalizableObject2 è un oggetto live e viene mantenuto nell'heap gestito.

Più oggetti morti

Cosa succede se due (o diversi) oggetti altrimenti morti si riferiscono l'un l'altro? Questo è mostrato nell'esempio seguente, supponendo che OtherObject sia una proprietà pubblica di FinalizableObject:

```
var obj1 = new FinalizableObject1();
var obj2 = new FinalizableObject2();
obj1.OtherObject = obj2;
obj2.OtherObject = obj1;
obj1 = null; // Program no longer references Finalizable1 instance
obj2 = null; // Program no longer references Finalizable2 instance
// But the two objects still reference each other
GC.Collect();
```

Questo produce il seguente risultato:

```
<namespace>.FinalizedObject1 initialized
<namespace>.FinalizedObject2 initialized
<namespace>.FinalizedObject1 finalized
<namespace>.FinalizedObject2 finalized
```

I due oggetti sono finalizzati e liberati dall'heap gestito malgrado il riferimento reciproco (poiché nessun altro riferimento esiste a nessuno di essi da un oggetto effettivamente in esecuzione).

Riferimenti deboli

I riferimenti deboli sono ... riferimenti, ad altri oggetti (noti anche come "target"), ma "deboli" in quanto non impediscono che tali oggetti vengano raccolti con garbage collection. In altre parole, i riferimenti deboli non vengono conteggiati quando Garbage Collector valuta gli oggetti come "live" o "dead".

Il seguente codice:

```
var weak = new WeakReference<FinalizableObject>(new FinalizableObject());
GC.Collect();
```

Produce l'output:

```
<namespace>.FinalizableObject initialized
<namespace>.FinalizableObject finalized
```

L'oggetto viene liberato dall'heap gestito nonostante venga fatto riferimento dalla variabile WeakReference (ancora nell'ambito quando è stato richiamato il Garbage Collector).

Conseguenza n. 1: in qualsiasi momento, non è sicuro assumere se una destinazione WeakReference è ancora allocata nell'heap gestito o meno.

Conseguenza 2: ogni volta che un programma deve accedere al target di una Weakreference, deve essere fornito il codice per entrambi i casi, del target ancora assegnato o meno. Il metodo per accedere al target è TryGetTarget:

```
var target = new object(); // Any object will do as target
var weak = new WeakReference<object>(target); // Create weak reference
target = null; // Drop strong reference to the target

// ... Many things may happen in-between

// Check whether the target is still available
if(weak.TryGetTarget(out target))
{
    // Use re-initialized target variable
    // To do whatever the target is needed for
}
else
{
    // Do something when there is no more target object
    // The target variable value should not be used here
}
```

La versione generica di WeakReference è disponibile da .Net 4.5. Tutte le versioni del framework forniscono una versione non generica, non tipizzata, costruita nello stesso modo e verificata come segue:

```
var target = new object(); // Any object will do as target
var weak = new WeakReference(target); // Create weak reference
target = null; // Drop strong reference to the target

// ... Many things may happen in-between

// Check whether the target is still available
if (weak.IsAlive)
{
    target = weak.Target;

    // Use re-initialized target variable
    // To do whatever the target is needed for
}
else
{
    // Do something when there is no more target object
    // The target variable value should not be used here
}
```

Dispose () rispetto ai finalizzatori

Implementa il metodo Dispose () (e dichiara la classe contenitore come IDisposable) come mezzo per garantire che tutte le risorse che pesano sulla memoria vengano liberate non appena l'oggetto non viene più utilizzato. Il "trucco" è che non vi è alcuna garanzia che il metodo Dispose () venga mai richiamato (a differenza dei finalizzatori che vengono sempre richiamati alla fine della vita

dell'oggetto).

Uno scenario è un programma che chiama Dispose () sugli oggetti che crea esplicitamente:

```
private void SomeFunction()
{
    // Initialize an object that uses heavy external resources
    var disposableObject = new ClassThatImplementsIDisposable();

    // ... Use that object

    // Dispose as soon as no longer used
    disposableObject.Dispose();

    // ... Do other stuff

    // The disposableObject variable gets out of scope here
    // The object will be finalized later on (no guarantee when)
    // But it no longer holds to the heavy external resource after it was disposed
}
```

Un altro scenario è la dichiarazione di una classe da istanziare dal framework. In questo caso la nuova classe solitamente eredita una classe base, ad esempio in MVC si crea una classe controller come sottoclasse di System.Web.Mvc.ControllerBase. Quando la classe base implementa l'interfaccia IDisposable, questo è un buon suggerimento che Dispose () sarebbe invocato correttamente dal framework - ma ancora una volta non c'è una forte garanzia.

Così Dispose () non è un sostituto di un finalizzatore; invece, i due dovrebbero essere usati per diversi scopi:

- Un finalizzatore alla fine libera risorse per evitare perdite di memoria che si verifichino diversamente
- Dispose () libera risorse (probabilmente le stesse) non appena queste non sono più necessarie, per alleggerire la pressione sull'assegnazione complessiva della memoria.

Smaltimento e finalizzazione adeguati degli oggetti

Dato che Dispose () ei finalizzatori sono finalizzati a scopi diversi, una classe che gestisce risorse di memoria esterne pesanti dovrebbe implementare entrambi. La conseguenza è scrivere la classe in modo che gestisca bene due possibili scenari:

- Quando viene invocato solo il finalizzatore
- Quando viene invocato Dispose () prima e successivamente viene anche chiamato il finalizzatore

Una soluzione è scrivere il codice cleanup in modo tale che eseguirlo una o due volte produca lo stesso risultato di una sola volta. La fattibilità dipende dalla natura della pulizia, ad esempio:

- La chiusura di una connessione al database già chiusa non avrebbe probabilmente alcun effetto, quindi funziona
- L'aggiornamento di alcuni "conteggi di utilizzo" è pericoloso e produrrebbe un risultato errato se chiamato due volte anziché una volta.

Una soluzione più sicura garantisce che il codice di pulizia venga chiamato una sola volta, una sola volta, indipendentemente dal contesto esterno. Questo può essere ottenuto con il "modo classico" usando una bandiera dedicata:

```
public class DisposableFinalizable1: IDisposable
{
    private bool disposed = false;

    ~DisposableFinalizable1() { Cleanup(); }

    public void Dispose() { Cleanup(); }

    private void Cleanup()
    {
        if(!disposed)
        {
            // Actual code to release resources gets here, then
            disposed = true;
        }
    }
}
```

In alternativa, Garbage Collector fornisce un metodo specifico SuppressFinalize () che consente di ignorare il finalizzatore dopo l'invocazione di Dispose:

```
public class DisposableFinalizable2 : IDisposable
{
    ~DisposableFinalizable2() { Cleanup(); }

    public void Dispose()
    {
        Cleanup();
        GC.SuppressFinalize(this);
    }

    private void Cleanup()
    {
        // Actual code to release resources gets here
    }
}
```

Leggi Raccolta dei rifiuti online: https://riptutorial.com/it/dot-net/topic/9636/raccolta-dei-rifiuti

Capitolo 38: ReadOnlyCollections

Osservazioni

Un `ReadOnlyCollection` fornisce una vista di sola lettura a una raccolta esistente (la 'collezione di origine').

Gli oggetti non vengono aggiunti o rimossi direttamente da `ReadOnlyCollection` . Invece, vengono aggiunti e rimossi dalla raccolta di origine e `ReadOnlyCollection` rifletterà queste modifiche all'origine.

Il numero e l'ordine degli elementi all'interno di `ReadOnlyCollection` non possono essere modificati, ma le proprietà degli elementi possono essere e i metodi possono essere chiamati, presupponendo che siano in ambito.

Utilizzare `ReadOnlyCollection` quando si desidera consentire al codice esterno di visualizzare la raccolta senza essere in grado di modificarla, ma è comunque possibile modificare la raccolta autonomamente.

Guarda anche

- `ObservableCollection<T>`
- `ReadOnlyObservableCollection<T>`

ReadOnlyCollections vs ImmutableCollection

A `ReadOnlyCollection` differenzia da `ImmutableCollection` in quanto non è possibile modificare un `ImmutableCollection` dopo averlo creato: esso contiene sempre `n` elementi e non possono essere sostituiti o riordinati. Una `ReadOnlyCollection` , d'altra parte, non può essere modificata direttamente, ma gli elementi possono ancora essere aggiunti / rimossi / riordinati utilizzando la raccolta di origine.

Examples

Creare un ReadOnlyCollection

Uso del Costruttore

Un `ReadOnlyCollection` viene creato passando un oggetto `IList` esistente nel costruttore:

```
var groceryList = new List<string> { "Apple", "Banana" };
var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);
```

Utilizzando LINQ

Inoltre, LINQ fornisce un metodo di estensione `AsReadOnly()` per oggetti `IList` :

```
var readOnlyVersion = groceryList.AsReadOnly();
```

Nota

In genere, si desidera mantenere la raccolta di origine in privato e consentire l'accesso pubblico a `ReadOnlyCollection` . Mentre è possibile creare un `ReadOnlyCollection` da un elenco in linea, non sarebbe possibile modificare la raccolta dopo averla creata.

```
var readOnlyGroceryList = new List<string> {"Apple", "Banana"}.AsReadOnly();
// Great, but you will not be able to update the grocery list because
//  you do not have a reference to the source list anymore!
```

Se ti accorgi di farlo, potresti prendere in considerazione l'utilizzo di un'altra struttura dati, come `ImmutableCollection` .

Aggiornamento di ReadOnlyCollection

Un `ReadOnlyCollection` non può essere modificato direttamente. Invece, la collezione di origine viene aggiornata e `ReadOnlyCollection` rifletterà queste modifiche. Questa è la caratteristica chiave di `ReadOnlyCollection` .

```
var groceryList = new List<string> { "Apple", "Banana" };

var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);

var itemCount = readOnlyGroceryList.Count;  // There are currently 2 items

//readOnlyGroceryList.Add("Candy");         // Compiler Error - Items cannot be added to a
ReadOnlyCollection object
groceryList.Add("Vitamins");                // ..but they can be added to the original
collection

itemCount = readOnlyGroceryList.Count;      // Now there are 3 items
var lastItem = readOnlyGroceryList.Last();  // The last item on the read only list is now
"Vitamins"
```

Visualizza la demo

Avvertenza: gli elementi di ReadOnlyCollection non sono intrinsecamente di sola lettura

Se la raccolta di origine è di un tipo non immutabile, è possibile modificare gli elementi accessibili tramite `ReadOnlyCollection` .

```
public class Item
{
    public string Name { get; set; }
    public decimal Price { get; set; }
```

```
}

public static void FillOrder()
{
    // An order is generated
    var order = new List<Item>
    {
        new Item { Name = "Apple", Price = 0.50m },
        new Item { Name = "Banana", Price = 0.75m },
        new Item { Name = "Vitamins", Price = 5.50m }
    };

    // The current sub total is $6.75
    var subTotal = order.Sum(item => item.Price);

    // Let the customer preview their order
    var customerPreview = new ReadOnlyCollection<Item>(order);

    // The customer can't add or remove items, but they can change
    //   the price of an item, even though it is a ReadOnlyCollection
    customerPreview.Last().Price = 0.25m;

    // The sub total is now only $1.50!
    subTotal = order.Sum(item => item.Price);
}
```

Visualizza la demo

Leggi ReadOnlyCollections online: https://riptutorial.com/it/dot-net/topic/6906/readonlycollections

Capitolo 39: Riflessione

Examples

Cos'è un assemblaggio?

Gli assiemi sono la base di qualsiasi applicazione Common Language Runtime (CLR) . Ogni tipo che definisci, insieme ai suoi metodi, proprietà e il loro bytecode, viene compilato e inserito in un Assembly.

```
using System.Reflection;
```

```
Assembly assembly = this.GetType().Assembly;
```

Gli assembly sono auto-documentanti: non solo contengono tipi, metodi e il loro codice IL, ma anche i metadati necessari per ispezionarli e consumarli, sia in fase di compilazione che in fase di runtime:

```
Assembly assembly = Assembly.GetExecutingAssembly();

foreach (var type in assembly.GetTypes())
{
    Console.WriteLine(type.FullName);
}
```

Gli assembly hanno nomi che descrivono la loro identità completa e univoca:

```
Console.WriteLine(typeof(int).Assembly.FullName);
// Will print: "mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

Se questo nome include un `PublicKeyToken` , viene chiamato un *nome* `PublicKeyToken` . Assegnare un nome forte a un assembly è il processo di creazione di una firma utilizzando la chiave privata corrispondente alla chiave pubblica distribuita con l'assembly. Questa firma viene aggiunta al manifest Assembly, che contiene i nomi e gli hash di tutti i file che compongono l'assembly e il suo `PublicKeyToken` diventa parte del nome. Gli assembly che hanno lo stesso nome forte dovrebbero essere identici; i nomi forti vengono utilizzati nel controllo delle versioni e per impedire conflitti di assembly.

Come creare un oggetto di T usando Reflection

Utilizzando il costruttore predefinito

```
T variable = Activator.CreateInstance(typeof(T));
```

Utilizzando il costruttore parametrizzato

```
T variable = Activator.CreateInstance(typeof(T), arg1, arg2);
```

Creazione di oggetti e impostazione delle proprietà mediante la riflessione

Diciamo che abbiamo una classe di `Classy` che ha proprietà Propertua

```
public class Classy
{
    public string Propertua {get; set;}
}
```

per impostare `Propertua` usando la riflessione:

```
var typeOfClassy = typeof (Classy);
var classy = new Classy();
var prop = typeOfClassy.GetProperty("Propertua");
prop.SetValue(classy, "Value");
```

Ottenere un attributo di enum con riflessione (e memorizzarlo nella cache)

Gli attributi possono essere utili per indicare i metadati sull'enumerazione. Ottenere il valore di questo può essere lento, quindi è importante memorizzare i risultati nella cache.

```
    private static Dictionary<object, object> attributeCache = new Dictionary<object,
object>();

    public static T GetAttribute<T, V>(this V value)
        where T : Attribute
        where V : struct
    {
        object temp;

        // Try to get the value from the static cache.
        if (attributeCache.TryGetValue(value, out temp))
        {
            return (T) temp;
        }
        else
        {
            // Get the type of the struct passed in.
            Type type = value.GetType();
            FieldInfo fieldInfo = type.GetField(value.ToString());

            // Get the custom attributes of the type desired found on the struct.
            T[] attribs = (T[])fieldInfo.GetCustomAttributes(typeof(T), false);

            // Return the first if there was a match.
            var result = attribs.Length > 0 ? attribs[0] : null;

            // Cache the result so future checks won't need reflection.
            attributeCache.Add(value, result);

            return result;
        }
    }
```

Confronta due oggetti con la riflessione

```
public class Equatable
{
    public string field1;

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        var type = obj.GetType();
        if (GetType() != type)
            return false;

        var fields = type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            if (field.GetValue(this) != field.GetValue(obj))
                return false;

        return true;
    }

    public override int GetHashCode()
    {
        var accumulator = 0;
        var fields = GetType().GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            accumulator = unchecked ((accumulator * 937) ^
field.GetValue(this).GetHashCode());

        return accumulator;
    }
}
```

Nota: questo esempio esegue una comparazione basata sul campo (ignora campi e proprietà statici) per semplicità

Leggi Riflessione online: https://riptutorial.com/it/dot-net/topic/44/riflessione

Capitolo 40: Scrivi e leggi dallo stream di StdErr

Examples

Scrivi su output errore standard utilizzando Console

```
var sourceFileName = "NonExistingFile";
try
{
    System.IO.File.Copy(sourceFileName, "DestinationFile");
}
catch (Exception e)
{
    var stdErr = Console.Error;
    stdErr.WriteLine($"Failed to copy '{sourceFileName}': {e.Message}");
}
```

Leggi da errore standard del processo figlio

```
var errors = new System.Text.StringBuilder();
var process = new Process
{
    StartInfo = new ProcessStartInfo
    {
        RedirectStandardError = true,
        FileName = "xcopy.exe",
        Arguments = "\"NonExistingFile\" \"DestinationFile\"",
        UseShellExecute = false
    },

};
process.ErrorDataReceived += (s, e) => errors.AppendLine(e.Data);
process.Start();
process.BeginErrorReadLine();
process.WaitForExit();

if (errors.Length > 0) // something went wrong
    System.Console.Error.WriteLine($"Child process error: \r\n {errors}");
```

Leggi Scrivi e leggi dallo stream di StdErr online: https://riptutorial.com/it/dot-net/topic/10779/scrivi-e-leggi-dallo-stream-di-stderr

Capitolo 41: Serializzazione JSON

Osservazioni

JavaScriptSerializer vs Json.NET

La classe `JavaScriptSerializer` è stata introdotta in .NET 3.5 ed è utilizzata internamente dal livello di comunicazione asincrona di .NET per le applicazioni abilitate AJAX. Può essere utilizzato per lavorare con JSON nel codice gestito.

Nonostante l'esistenza della classe `JavaScriptSerializer` , Microsoft consiglia di utilizzare la libreria Json.NET open source per la serializzazione e la deserializzazione. Json.NET offre prestazioni migliori e un'interfaccia più amichevole per mappare JSON a classi personalizzate (sarebbe necessario un oggetto `JavaScriptConverter` personalizzato per ottenere lo stesso risultato con `JavaScriptSerializer`).

Examples

Deserializzazione tramite System.Web.Script.Serialization.JavaScriptSerializer

Il metodo `JavaScriptSerializer.Deserialize<T>(input)` tenta di deserializzare una stringa di JSON valido in un oggetto del tipo specificato `<T>` , utilizzando i mapping predefiniti supportati nativamente da `JavaScriptSerializer` .

```
using System.Collections;
using System.Web.Script.Serialization;

// ...

string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

JavaScriptSerializer JSS = new JavaScriptSerializer();
Dictionary<string, object> parsedObj = JSS.Deserialize<Dictionary<string, object>>(rawJSON);

string name = parsedObj["Name"].toString();
ArrayList numbers = (ArrayList)parsedObj["Numbers"]
```

Nota: l'oggetto `JavaScriptSerializer` stato introdotto in .NET versione 3.5

Deserializzazione con Json.NET

```
internal class Sequence{
    public string Name;
    public List<int> Numbers;
}

// ...

string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
```

```
Sequence sequence = JsonConvert.DeserializeObject<Sequence>(rawJSON);
```

Per ulteriori informazioni, consultare il sito ufficiale Json.NET .

Nota: Json.NET supporta .NET versione 2 e successive.

Serializzazione con Json.NET

```
[JsonObject("person")]
public class Person
{
    [JsonProperty("name")]
    public string PersonName { get; set; }
    [JsonProperty("age")]
    public int PersonAge { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

Person person = new Person { PersonName = "Andrius", PersonAge = 99, Address = "Some address"
};
string rawJson = JsonConvert.SerializeObject(person);

Console.WriteLine(rawJson); // {"name":"Andrius","age":99}
```

Si noti come le proprietà (e le classi) possono essere decorate con attributi per cambiare il loro aspetto nella stringa json risultante o per rimuoverli da json string a tutti (JsonIgnore).

Ulteriori informazioni sugli attributi di serializzazione Json.NET possono essere trovate qui .

In C #, gli identificatori pubblici sono scritti in *PascalCase* per convenzione. In JSON, la convenzione è usare *CamelCase* per tutti i nomi. È possibile utilizzare un resolver di contratto per convertire tra i due.

```
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

public void ToJson() {
    Person person = new Person { Name = "Andrius", Age = 99, Address = "Some address" };
    var resolver = new CamelCasePropertyNamesContractResolver();
    var settings = new JsonSerializerSettings { ContractResolver = resolver };
    string json = JsonConvert.SerializeObject(person, settings);

    Console.WriteLine(json); // {"name":"Andrius","age":99}
}
```

Serializzazione-Deserializzazione usando Newtonsoft.Json

A differenza degli altri helper, questo usa gli helper di classe statici per serializzare e deserializzare, quindi è un po 'più semplice degli altri da usare.

```
using Newtonsoft.Json;

var rawJSON      = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
var fibo         = JsonConvert.DeserializeObject<Dictionary<string, object>>(rawJSON);
var rawJSON2     = JsonConvert.SerializeObject(fibo);
```

Associazione dinamica

Json.NET di Newtonsoft ti permette di legare json in modo dinamico (usando ExpandoObject / oggetti dinamici) senza la necessità di creare il tipo esplicitamente.

serializzazione

```
dynamic jsonObject = new ExpandoObject();
jsonObject.Title   = "Merchent of Venice";
jsonObject.Author  = "William Shakespeare";
Console.WriteLine(JsonConvert.SerializeObject(jsonObject));
```

De-serializzazione

```
var rawJson = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
dynamic parsedJson = JObject.Parse(rawJson);
Console.WriteLine("Name: " + parsedJson.Name);
Console.WriteLine("Name: " + parsedJson.Numbers.Length);
```

Si noti che le chiavi nell'oggetto rawJson sono state trasformate in variabili membro nell'oggetto dinamico.

Ciò è utile nei casi in cui un'applicazione può accettare / produrre diversi formati di JSON. Si consiglia comunque di utilizzare un ulteriore livello di convalida per la stringa Json o per l'oggetto dinamico generato come risultato della serializzazione / de-serializzazione.

Serializzazione tramite Json.NET con JsonSerializerSettings

Questo serializzatore ha alcune caratteristiche interessanti che il serializzatore json predefinito non ha, come la gestione del valore Null, devi solo creare `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
NullValueHandling = NullValueHandling.Ignore});
   return result;
}
```

Un altro problema serio di serializzazione in .net è il loop autoreferenziale. Nel caso di uno

studente iscritto a un corso, la sua istanza ha una proprietà del corso e un corso ha una raccolta di studenti che significa una `List<Student>` che creerà un ciclo di riferimento. Puoi gestirlo con `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```

Puoi mettere varie opzioni di serializzazione come questa:

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
NullValueHandling = NullValueHandling.Ignore, ReferenceLoopHandling =
ReferenceLoopHandling.Ignore});
   return result;
}
```

Leggi Serializzazione JSON online: https://riptutorial.com/it/dot-net/topic/183/serializzazione-json

Capitolo 42: Server HTTP

Examples

File server HTTP di sola lettura di base (HttpListener)

Gli appunti:

Questo esempio deve essere eseguito in modalità amministrativa.

È supportato solo un client simultaneo.

Per semplicità, si presume che i *nomi* dei *file* siano tutti ASCII (per la parte *nome file* nell'intestazione *Content-Disposition*) e che gli errori di accesso ai file non vengano gestiti.

```
using System;
using System.IO;
using System.Net;

class HttpFileServer
{
    private static HttpListenerResponse response;
    private static HttpListener listener;
    private static string baseFilesystemPath;

    static void Main(string[] args)
    {
        if (!HttpListener.IsSupported)
        {
            Console.WriteLine(
                "*** HttpListener requires at least Windows XP SP2 or Windows Server 2003.");
            return;
        }

        if(args.Length < 2)
        {
            Console.WriteLine("Basic read-only HTTP file server");
            Console.WriteLine();
            Console.WriteLine("Usage: httpfileserver <base filesystem path> <port>");
            Console.WriteLine("Request format: http://url:port/path/to/file.ext");
            return;
        }

        baseFilesystemPath = Path.GetFullPath(args[0]);
        var port = int.Parse(args[1]);

        listener = new HttpListener();
        listener.Prefixes.Add("http://*:" + port + "/");
        listener.Start();

        Console.WriteLine("--- Server stated, base path is: " + baseFilesystemPath);
        Console.WriteLine("--- Listening, exit with Ctrl-C");
        try
        {
            ServerLoop();
```

```
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex);
            if(response != null)
            {
                SendErrorResponse(500, "Internal server error");
            }
        }
    }

    static void ServerLoop()
    {
        while(true)
        {
            var context = listener.GetContext();

            var request = context.Request;
            response = context.Response;
            var fileName = request.RawUrl.Substring(1);
            Console.WriteLine(
                "--- Got {0} request for: {1}",
                request.HttpMethod, fileName);

            if (request.HttpMethod.ToUpper() != "GET")
            {
                SendErrorResponse(405, "Method must be GET");
                continue;
            }

            var fullFilePath = Path.Combine(baseFilesystemPath, fileName);
            if(!File.Exists(fullFilePath))
            {
                SendErrorResponse(404, "File not found");
                continue;
            }

            Console.Write("    Sending file...");
            using (var fileStream = File.OpenRead(fullFilePath))
            {
                response.ContentType = "application/octet-stream";
                response.ContentLength64 = (new FileInfo(fullFilePath)).Length;
                response.AddHeader(
                    "Content-Disposition",
                    "Attachment; filename=\"" + Path.GetFileName(fullFilePath) + "\"");
                fileStream.CopyTo(response.OutputStream);
            }

            response.OutputStream.Close();
            response = null;
            Console.WriteLine(" Ok!");
        }
    }

    static void SendErrorResponse(int statusCode, string statusResponse)
    {
        response.ContentLength64 = 0;
        response.StatusCode = statusCode;
        response.StatusDescription = statusResponse;
        response.OutputStream.Close();
        Console.WriteLine("*** Sent error: {0} {1}", statusCode, statusResponse);
```

```
    }
}
```

File server HTTP di sola lettura di base (ASP.NET Core)

1 - Crea una cartella vuota, conterrà i file creati nei passaggi successivi.

2 - Creare un file denominato `project.json` con il seguente contenuto (regolare il numero di porta e `rootDirectory` come appropriato):

```
{
  "dependencies": {
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel --server.urls http://localhost:60000"
  },

  "frameworks": {
    "dnxcore50": { }
  },

  "fileServer": {
    "rootDirectory": "c:\\users\\username\\Documents"
  }
}
```

3 - Creare un file denominato `Startup.cs` con il seguente codice:

```
using System;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.FileProviders;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.StaticFiles;
using Microsoft.Extensions.Configuration;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        var builder = new ConfigurationBuilder();
        builder.AddJsonFile("project.json");
        var config = builder.Build();
        var rootDirectory = config["fileServer:rootDirectory"];
        Console.WriteLine("File server root directory: " + rootDirectory);

        var fileProvider = new PhysicalFileProvider(rootDirectory);

        var options = new StaticFileOptions();
        options.ServeUnknownFileTypes = true;
        options.FileProvider = fileProvider;
        options.OnPrepareResponse = context =>
        {
            context.Context.Response.ContentType = "application/octet-stream";
            context.Context.Response.Headers.Add(
```

```
                "Content-Disposition",
                $"Attachment; filename=\"{context.File.Name}\"");
        };

        app.UseStaticFiles(options);
    }
}
```

4 - Aprire un prompt dei comandi, navigare fino alla cartella ed eseguire:

```
dnvm use 1.0.0-rc1-final -r coreclr -p
dnu restore
```

Nota: questi comandi devono essere eseguiti una sola volta. Utilizzare `dnvm list` per verificare il numero effettivo dell'ultima versione installata del CLR principale.

5 - Avviare il server con: `dnx web` . I file possono ora essere richiesti su `http://localhost:60000/path/to/file.ext` .

Per semplicità, si presume che i nomi dei file siano tutti ASCII (per la parte nome file nell'intestazione Content-Disposition) e che gli errori di accesso ai file non vengano gestiti.

Leggi Server HTTP online: https://riptutorial.com/it/dot-net/topic/53/server-http

Capitolo 43: Sistema di imballaggio NuGet

Osservazioni

NuGet.org :

> NuGet è il gestore di pacchetti per la piattaforma di sviluppo Microsoft, tra cui .NET. Gli strumenti client di NuGet offrono la possibilità di produrre e consumare pacchetti. La Galleria NuGet è il repository centrale di pacchetti utilizzato da tutti gli autori e consumatori di pacchetti.

Immagini negli esempi per gentile concessione di NuGet.org .

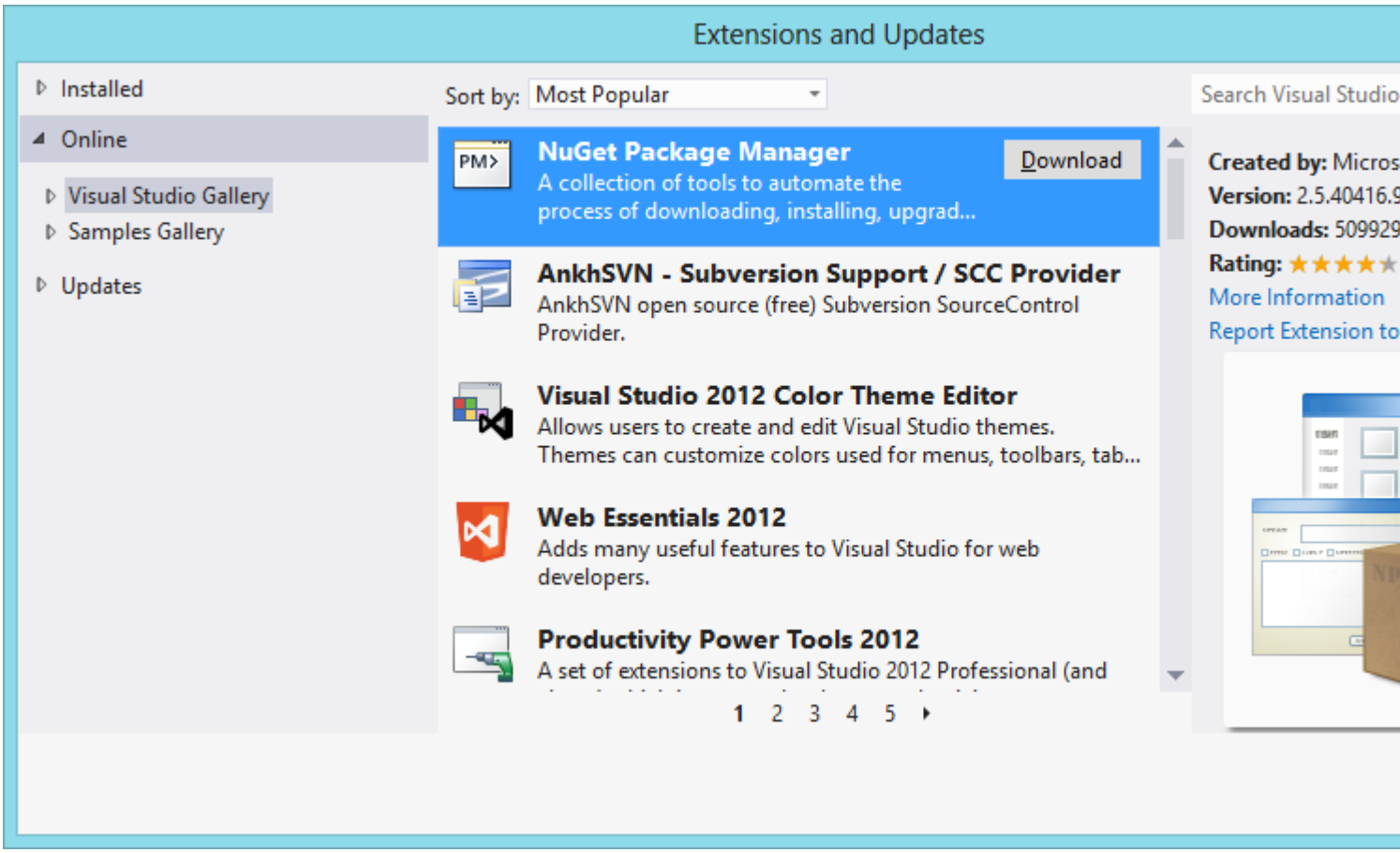
Examples

Installazione di NuGet Package Manager

Per poter gestire i pacchetti dei progetti, è necessario il Gestore pacchetti NuGet. Questa è un'estensione di Visual Studio, spiegata nei documenti ufficiali: Installazione e aggiornamento di NuGet Client .

A partire da Visual Studio 2012, NuGet è incluso in ogni edizione e può essere utilizzato da: Strumenti -> Gestore pacchetti NuGet -> Console Gestione pacchetti.

Lo fai attraverso il menu Strumenti di Visual Studio, facendo clic su Estensioni e aggiornamenti:



Questo installa sia la GUI:

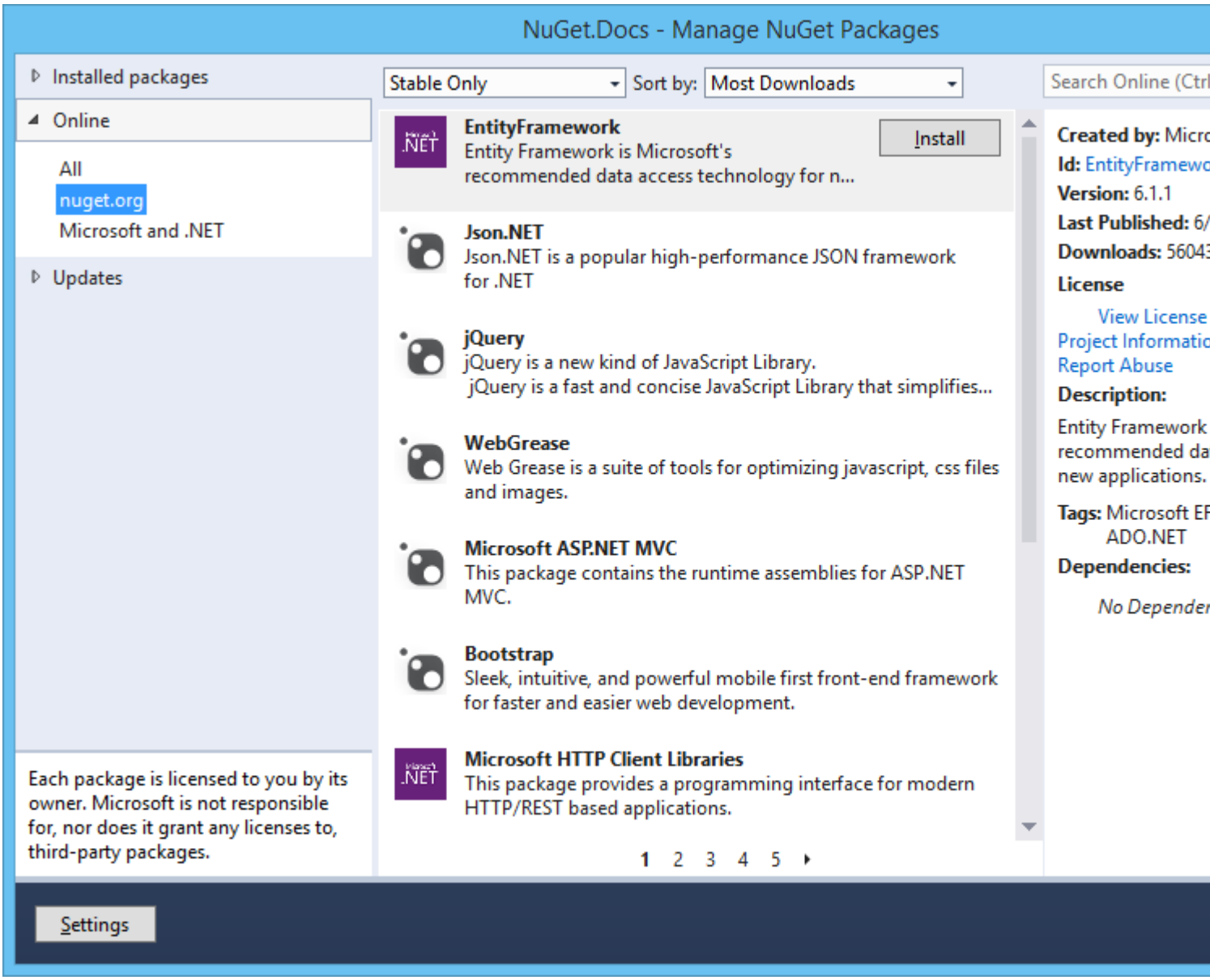
- Disponibile facendo clic su "Gestisci pacchetti NuGet ..." su un progetto o sulla sua cartella Riferimenti

E la console di Gestione pacchetti:

- Strumenti -> NuGet Package Manager -> Console Gestione pacchetti.

Gestione dei pacchetti tramite l'interfaccia utente

Quando fai clic con il pulsante destro del mouse su un progetto (o sulla sua cartella Riferimenti), puoi fare clic sull'opzione "Gestisci pacchetti NuGet ...". Questo mostra la finestra di dialogo Gestore pacchetti .



Gestione dei pacchetti tramite la console

Fai clic sui menu Strumenti -> NuGet Package Manager -> Gestione pacchetti manager per mostrare la console nel tuo IDE. Documentazione ufficiale qui .

Qui puoi rilasciare, tra gli altri, i comandi `install-package` che installano il pacchetto inserito nel "Progetto predefinito" attualmente selezionato:

```
Install-Package Elmah
```

Puoi anche fornire il progetto per installare il pacchetto su, ignorando il progetto selezionato nel menu a discesa "Progetto predefinito":

```
Install-Package Elmah -ProjectName MyFirstWebsite
```

Aggiornamento di un pacchetto

Per aggiornare un pacchetto usa il seguente comando:

```
PM> Update-Package EntityFramework
```

dove EntityFramework è il nome del pacchetto da aggiornare. Si noti che l'aggiornamento verrà eseguito per tutti i progetti e quindi è diverso da `Install-Package EntityFramework` che si installerà solo in "Progetto predefinito".

Puoi anche specificare un singolo progetto in modo esplicito:

```
PM> Update-Package EntityFramework -ProjectName MyFirstWebsite
```

Disinstallazione di un pacchetto

```
PM> Uninstall-Package EntityFramework
```

Disinstallazione di un pacchetto da un progetto in una soluzione

```
PM> Uninstall-Package -ProjectName MyProjectB EntityFramework
```

Installazione di una versione specifica di un pacchetto

```
PM> Install-Package EntityFramework -Version 6.1.2
```

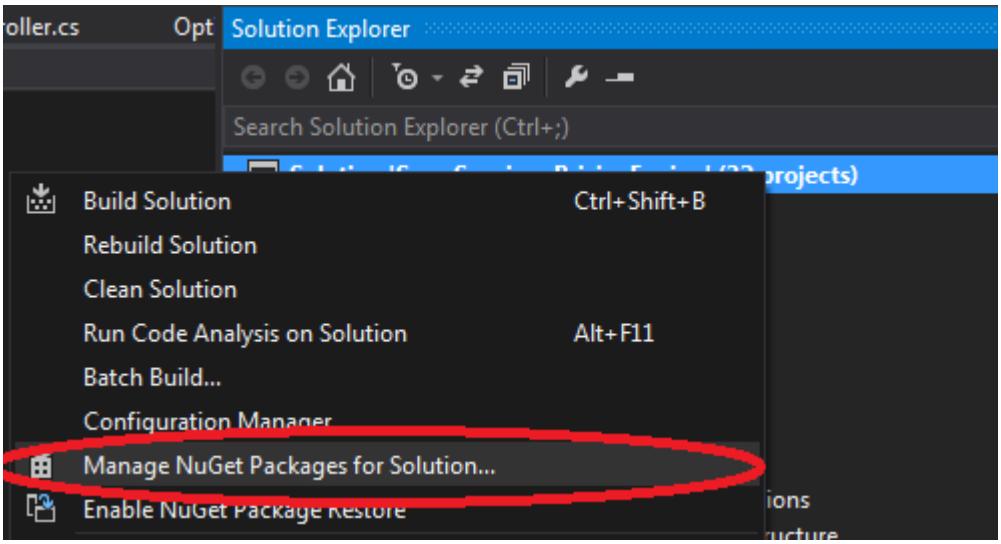
Aggiunta di un feed sorgente del pacchetto (MyGet, Klondike, ect)

```
nuget sources add -name feedname -source http://sourcefeedurl
```

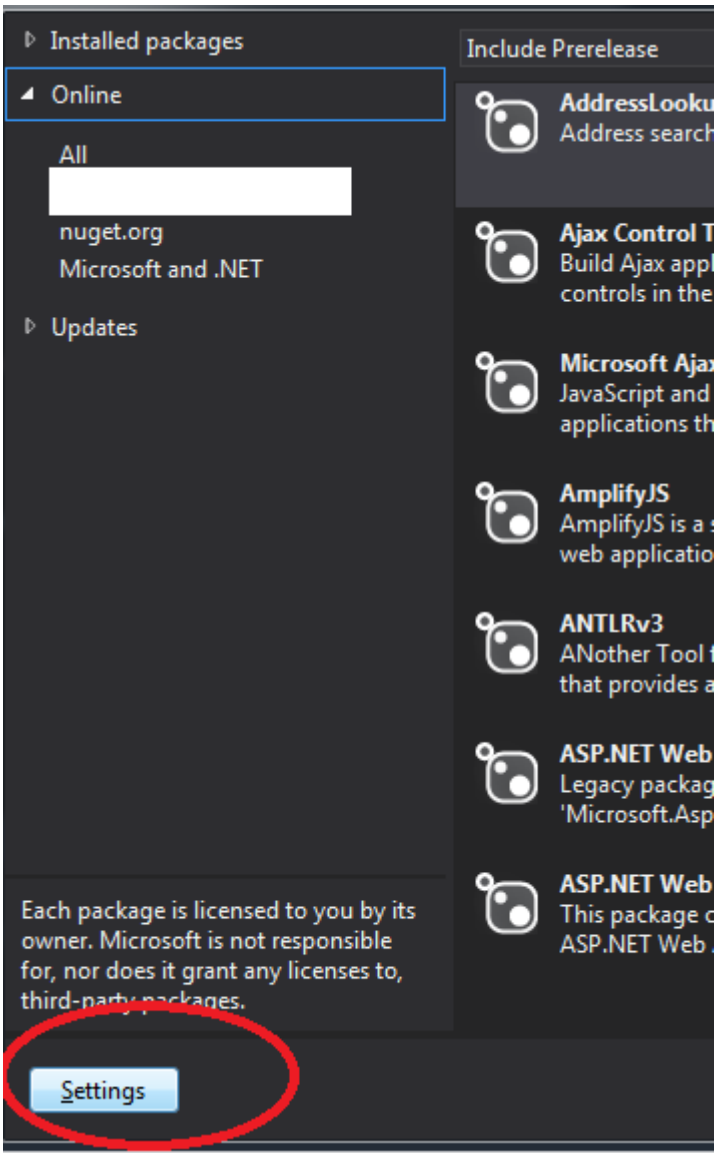
Usando diverse fonti di pacchetti Nuget (locali) usando l'interfaccia utente

È normale che la società configuri il proprio server nuget per la distribuzione dei pacchetti tra diversi team.

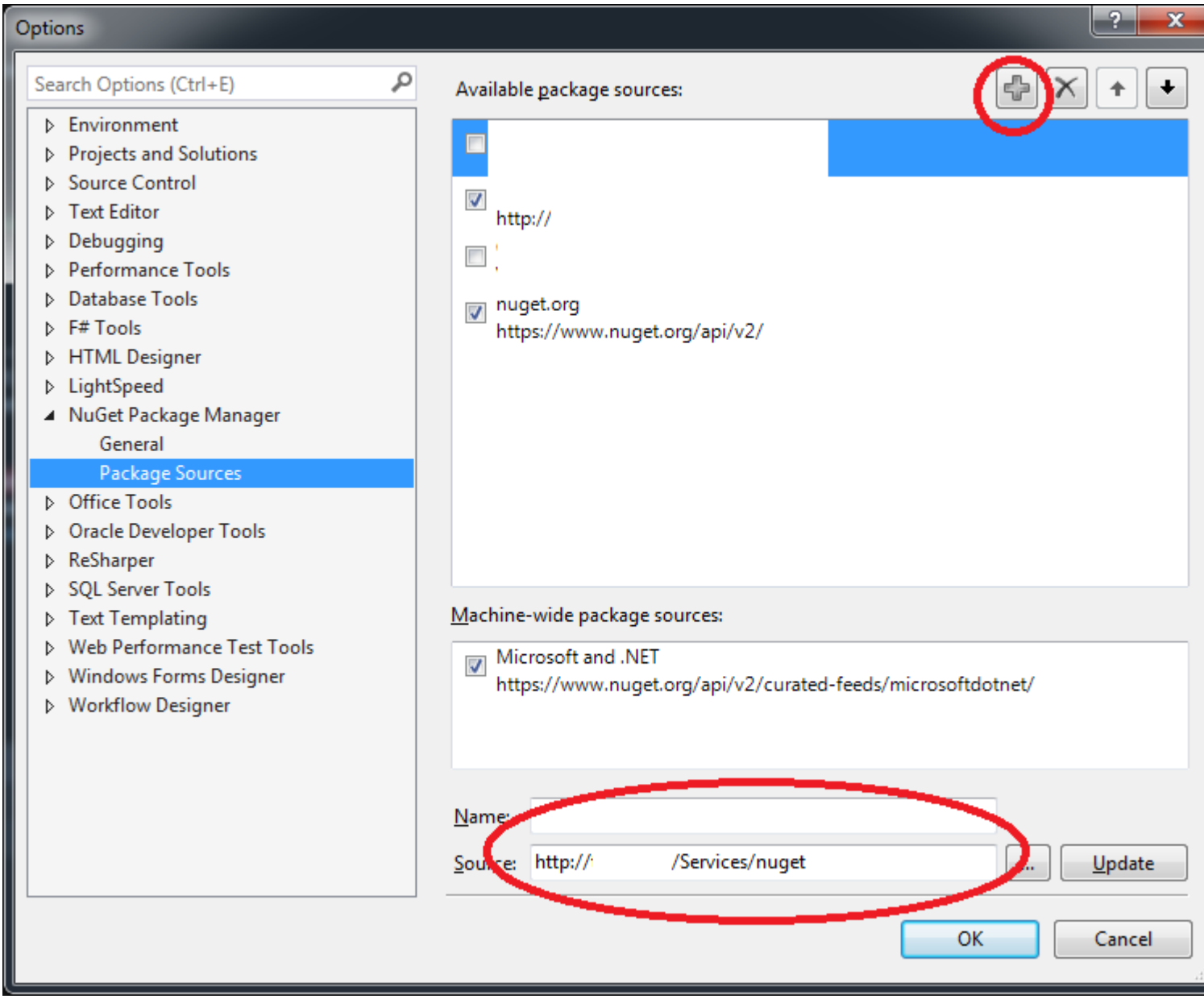
1. Vai a Solution Explorer e fai clic sul pulsante `destro del mouse` , quindi scegli `Manage NuGet Packages for Solution`



2. Nella finestra che si apre fare clic su `Settings`



3. Fai clic su `+` nell'angolo in alto a destra, quindi aggiungi il nome e l'URL che punta al tuo server nuget locale.



disinstallare una versione specifica del pacchetto

```
PM> uninstall-Package EntityFramework -Version 6.1.2
```

Leggi Sistema di imballaggio NuGet online: https://riptutorial.com/it/dot-net/topic/43/sistema-di-imballaggio-nuget

Capitolo 44: SpeechRecognitionEngine classe per riconoscere il parlato

Sintassi

- SpeechRecognitionEngine ()
- SpeechRecognitionEngine.LoadGrammar (Grammatica grammatica)
- SpeechRecognitionEngine.SetInputToDefaultAudioDevice ()
- SpeechRecognitionEngine.RecognizeAsync (modalità RecognizeMode)
- GrammarBuilder ()
- GrammarBuilder.Append (Scelte scelte)
- Scelte (parametri stringa [] scelte)
- Grammatica (costruttore di GrammarBuilder)

Parametri

`LoadGrammar` : parametri	Dettagli
grammatica	La grammatica da caricare. Ad esempio, un oggetto `DictationGrammar` per consentire la dettatura di testo libero.
`RecognizeAsync` : Parameters	**Dettagli**
modalità	`RecognizeMode` per il riconoscimento corrente: `Single` per un solo riconoscimento, `Multiple` per consentire multipli.
`GrammarBuilder.Append` : Parameters	**Dettagli**
scelte	Aggiunge alcune scelte al generatore di grammatica. Ciò significa che, quando l'utente digita la parola, il riconoscitore può seguire "rami" diversi da una grammatica.
Costruttore di `Choices` : parametri	**Dettagli**
scelte	Una serie di scelte per il generatore di grammatica. Vedi `GrammarBuilder.Append` .
Costruttore di `Grammar` : parametro	**Dettagli**
costruttore	Il `GrammarBuilder` per costruire una `Grammar` da.

Osservazioni

Per utilizzare `SpeechRecognitionEngine` , la versione di Windows deve avere il riconoscimento vocale abilitato.

È necessario aggiungere un riferimento a `System.Speech.dll` prima di poter utilizzare le classi del linguaggio.

Examples

Riconoscimento asincrono del parlato per dettatura di testo libero

```
using System.Speech.Recognition;

// ...

SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
recognitionEngine.LoadGrammar(new DictationGrammar());
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
    Console.WriteLine("You said: {0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

Riconoscimento asincrono del parlato basato su un insieme limitato di frasi

```
SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
GrammarBuilder builder = new GrammarBuilder();
builder.Append(new Choices("I am", "You are", "He is", "She is", "We are", "They are"));
builder.Append(new Choices("friendly", "unfriendly"));
recognitionEngine.LoadGrammar(new Grammar(builder));
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
    Console.WriteLine("You said: {0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

Leggi SpeechRecognitionEngine classe per riconoscere il parlato online:
https://riptutorial.com/it/dot-net/topic/69/speechrecognitionengine-classe-per-riconoscere-il-parlato

Capitolo 45: Stack e Heap

Osservazioni

Vale la pena notare che nel dichiarare un tipo di riferimento, il suo valore iniziale sarà `null` .
Questo perché non punta ancora a una posizione in memoria, ed è uno stato perfettamente valido.
Tuttavia, con l'eccezione dei tipi nullable, i tipi di valore devono sempre avere sempre un valore.

Examples

Tipi di valore in uso

I tipi di valore contengono semplicemente un ***valore*** .

Tutti i tipi di valore derivano dalla classe System.ValueType e questo include la maggior parte dei tipi incorporati.

Quando si crea un nuovo tipo di valore, viene utilizzata un'area di memoria chiamata ***stack*** .
Lo stack crescerà di conseguenza, in base alla dimensione del tipo dichiarato. Ad esempio, ad un int verranno sempre assegnati 32 bit di memoria nello stack. Quando il tipo di valore non è più nell'ambito, lo spazio nello stack sarà deallocato.

Il codice seguente mostra un tipo di valore assegnato a una nuova variabile. Una struct viene utilizzata come un modo conveniente per creare un tipo di valore personalizzato (la classe System.ValueType non può essere estesa in altro modo).

La cosa importante da capire è che quando si assegna un tipo di valore, il valore stesso viene ***copiato*** nella nuova variabile, nel senso che abbiamo due istanze distinte dell'oggetto, che non possono influenzarsi a vicenda.

```
struct PersonAsValueType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsValueType personA;

        personA.Name = "Bob";

        var personB = personA;

        personA.Name = "Linda";

        Console.WriteLine(                // Outputs 'False' - because
```

```
            object.ReferenceEquals(       // personA and personB are referencing
                personA,                  // different areas of memory
                personB));

        Console.WriteLine(personA.Name);  // Outputs 'Linda'
        Console.WriteLine(personB.Name);  // Outputs 'Bob'
    }
}
```

Tipi di riferimento in uso

I tipi di riferimento comprendono sia un ***riferimento*** a un'area di memoria, sia un ***valore*** memorizzato all'interno di quell'area.
Questo è analogo ai puntatori in C / C ++.

Tutti i tipi di riferimento sono memorizzati su ciò che è noto come ***heap*** .
L'heap è semplicemente un'area di memoria gestita in cui sono memorizzati gli oggetti. Quando un nuovo oggetto viene istanziato, una parte dell'heap verrà allocata per l'utilizzo da quell'oggetto e verrà restituito un riferimento a tale posizione dell'heap. L'heap è gestito e gestito dal *garbage collector* e non consente l'intervento manuale.

Oltre allo spazio di memoria richiesto per l'istanza stessa, è necessario ulteriore spazio per memorizzare il riferimento stesso, insieme a ulteriori informazioni temporanee richieste da .NET CLR.

Il codice seguente mostra un tipo di riferimento assegnato a una nuova variabile. In questo caso, stiamo usando una classe, tutte le classi sono tipi di riferimento (anche se statici).

Quando un tipo di riferimento è assegnato a un'altra variabile, è il ***riferimento*** all'oggetto che viene copiato, **non** il valore stesso. Questa è una distinzione importante tra tipi di valore e tipi di riferimento.

Le implicazioni di questo sono che ora abbiamo *due* riferimenti allo stesso oggetto.
Qualsiasi modifica ai valori all'interno di quell'oggetto sarà riflessa da entrambe le variabili.

```
class PersonAsReferenceType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsReferenceType personA;

        personA = new PersonAsReferenceType { Name = "Bob" };

        var personB = personA;

        personA.Name = "Linda";

        Console.WriteLine(              // Outputs 'True' - because
```

```
            object.ReferenceEquals(      // personA and personB are referencing
                personA,                 // the *same* memory location
                personB));

        Console.WriteLine(personA.Name); // Outputs 'Linda'
        Console.WriteLine(personB.Name); // Outputs 'Linda'
    }
```

Leggi Stack e Heap online: https://riptutorial.com/it/dot-net/topic/9358/stack-e-heap

Capitolo 46: stringhe

Osservazioni

Nelle stringhe .NET `System.String` sono sequenze di caratteri `System.Char` , ogni carattere è un'unità di codice codificata UTF-16. Questa distinzione è importante perché la definizione del *linguaggio parlata* di *carattere* e la definizione di *carattere* .NET (e molte altre lingue) sono diverse.

Un *carattere* , che dovrebbe essere correttamente chiamato grafema , viene visualizzato come un glifo ed è definito da uno o più punti di codice Unicode. Ogni punto di codice è quindi codificato in una sequenza di unità di codice . Ora dovrebbe essere chiaro il motivo per cui un singolo `System.Char` non rappresenta sempre un grapheme, vediamo nel mondo reale come sono diversi:

- Un grafema, a causa della combinazione di caratteri , può risultare in due o più punti di codice: `à` è composto da due punti di codice: *U + 0061 LATIN LETTER A* e *U + 0300 LIVIN COMBINING GRAVE ACCENT* . Questo è l'errore più comune perché `"à ".Length == 2` mentre ci si può aspettare `1` .
- Ci sono caratteri duplicati, ad esempio `à` può essere un singolo punto di codice *U + 00E0 LATIN SMALL LETTER A WITH GRAVE* o due code-point come spiegato sopra. Ovviamente devono confrontare lo stesso: `"\u00e0" == "\u0061\u0300"` (anche se `"\u00e0".Length != "\u0061\u0300".Length`). Ciò è possibile a causa della *normalizzazione* delle *stringhe* eseguita dal metodo `String.Normalize()` .
- Una sequenza Unicode può contenere una sequenza composta o scomposta, per esempio il carattere □ *U + D55C HAN CHARACTER* può essere un singolo punto di codice (codificato come una singola unità di codice in UTF-16) o una sequenza decomposta delle sue sillabe □ , □ e □ . Devono essere paragonati allo stesso modo.
- Un punto di codice può essere codificato su più di una unità di codice: carattere □ *U + 2008A HAN CHARACTER* è codificato come due `System.Char` (`"\ud840\udc8a"`) anche se è solo un punto di codice: UTF-16 la codifica non è una dimensione fissa! Questa è una fonte di innumerevoli bachi (anche gravi bug di sicurezza), se per esempio la tua applicazione applica una lunghezza massima e ciecamente tronca una stringa in quel momento, puoi creare una stringa non valida.
- Alcune lingue hanno digraph e trigrammi, per esempio in `ch` ceco è una lettera standalone (dopo le `ore` e prima che `io` poi quando si ordina una lista di stringhe si dovrà *fyzika* prima *Chemie.*

Ci sono molti più problemi sulla gestione del testo, vedi per esempio Come posso eseguire un confronto con caratteri Unicode per confronto di caratteri? per un'introduzione più ampia e più collegamenti a argomenti correlati.

In generale, quando si tratta di testo *internazionale* , è possibile utilizzare questa semplice funzione per enumerare gli elementi di testo in una stringa (evitando di interrompere i surrogati e la codifica Unicode):

```
public static class StringExtensions
{
```

```
    public static IEnumerable<string> EnumerateCharacters(this string s)
    {
        if (s == null)
            return Enumerable.Empty<string>();

        var enumerator = StringInfo.GetTextElementEnumerator(s.Normalize());
        while (enumerator.MoveNext())
            yield return (string)enumerator.Value;
    }
}
```

Examples

Conta personaggi distinti

Se hai bisogno di contare caratteri distinti, per le ragioni spiegate nella sezione *Commenti* , non puoi semplicemente usare la proprietà `Length` perché è la lunghezza dell'array di `System.Char` che non sono caratteri ma code-unità (non code point Unicode) né grafemi). Se, ad esempio, scrivi semplicemente `text.Distinct().Count()` otterrai risultati errati, codice corretto:

```
int distinctCharactersCount = text.EnumerateCharacters().Count();
```

Un passo ulteriore è **contare le occorrenze di ogni carattere** , se le prestazioni non sono un problema, puoi semplicemente farlo in questo modo (in questo esempio, indipendentemente dal caso):

```
var frequencies = text.EnumerateCharacters()
    .GroupBy(x => x, StringComparer.CurrentCultureIgnoreCase)
    .Select(x => new { Character = x.Key, Count = x.Count() };
```

Conta personaggi

Se hai bisogno di contare i *caratteri* , per le ragioni spiegate nella sezione *Commenti* , non puoi semplicemente usare la proprietà Length perché è la lunghezza dell'array di `System.Char` che non sono caratteri ma code-unità (non code-points Unicode, né grafemi). Il codice corretto è quindi:

```
int length = text.EnumerateCharacters().Count();
```

Una piccola ottimizzazione può riscrivere il metodo di estensione `EnumerateCharacters()` appositamente per questo scopo:

```
public static class StringExtensions
{
    public static int CountCharacters(this string text)
    {
        if (String.IsNullOrEmpty(text))
            return 0;

        int count = 0;
        var enumerator = StringInfo.GetTextElementEnumerator(text);
```

```
        while (enumerator.MoveNext())
            ++count;

        return count;
    }
}
```

Conta le occorrenze di un personaggio

A causa delle ragioni spiegate nella sezione *Commenti* , non è possibile farlo semplicemente (a meno che non si vogliano contare le occorrenze di una specifica unità di codice):

```
int count = text.Count(x => x == ch);
```

Hai bisogno di una funzione più complessa:

```
public static int CountOccurrencesOf(this string text, string character)
{
    return text.EnumerateCharacters()
        .Count(x => String.Equals(x, character, StringComparer.CurrentCulture));
}
```

Si noti che il confronto delle stringhe (a differenza del confronto dei caratteri, che è invariante di cultura) deve sempre essere eseguito secondo le regole di una cultura specifica.

Dividere la stringa in blocchi di lunghezza fissa

Non possiamo spezzare una stringa in punti arbitrari (perché un `System.Char` può non essere valido solo perché è un personaggio che combina o parte di un surrogato) quindi il codice deve tenerne conto (notare che con *lunghezza* intendo il numero di *grafemi* non il numero di *unità* di *codice*):

```
public static IEnumerable<string> Split(this string value, int desiredLength)
{
    var characters = StringInfo.GetTextElementEnumerator(value);
    while (characters.MoveNext())
        yield return String.Concat(Take(characters, desiredLength));
}

private static IEnumerable<string> Take(TextElementEnumerator enumerator, int count)
{
    for (int i = 0; i < count; ++i)
    {
        yield return (string)enumerator.Current;

        if (!enumerator.MoveNext())
            yield break;
    }
}
```

Converti una stringa in / da un'altra codifica

Le stringhe .NET contengono `System.Char` (unità di codice UTF-16). Se vuoi salvare (o gestire) il

testo con un'altra codifica, devi lavorare con un array di `System.Byte` .

Le conversioni vengono eseguite da classi derivate da `System.Text.Encoder` e `System.Text.Decoder` che, insieme, possono convertire in / da un'altra codifica (da un byte *X* byte codificato in `byte[]` a un `System.String` e vice codificati in UTF-16 -versa).

Poiché il codificatore / decodificatore di solito funziona molto vicino tra loro, vengono raggruppati in una classe derivata da `System.Text.Encoding` , le classi derivate offrono conversioni alle / dalle codifiche più comuni (UTF-8, UTF-16 e così via).

Esempi:

Convertire una stringa in UTF-8

```
byte[] data = Encoding.UTF8.GetBytes("This is my text");
```

Converti dati UTF-8 in una stringa

```
var text = Encoding.UTF8.GetString(data);
```

Cambia la codifica di un file di testo esistente

Questo codice leggerà il contenuto di un file di testo con codifica UTF-8 e lo salverà codificato come UTF-16. Nota che questo codice non è ottimale se il file è grande perché leggerà tutto il suo contenuto in memoria:

```
var content = File.ReadAllText(path, Encoding.UTF8);
File.WriteAllText(content, Encoding.UTF16);
```

Object.ToString () metodo virtuale

Tutto in .NET è un oggetto, quindi ogni tipo ha il metodo `ToString()` definito nella classe `Object` che può essere sovrascritto. L'implementazione predefinita di questo metodo restituisce semplicemente il nome del tipo:

```
public class Foo
{
}

var foo = new Foo();
Console.WriteLine(foo); // outputs Foo
```

`ToString()` viene chiamato implicitamente quando si concatena il valore con una stringa:

```
public class Foo
{
    public override string ToString()
    {
        return "I am Foo";
    }
}

var foo = new Foo();
Console.WriteLine("I am bar and "+foo);// outputs I am bar and I am Foo
```

Il risultato di questo metodo è anche ampiamente utilizzato dagli strumenti di debug. Se, per qualche motivo, non si vuole sovrascrivere questo metodo, ma si desidera personalizzare il modo in cui il debugger mostra il valore del proprio tipo, utilizzare DebuggerDisplay Attribute (MSDN):

```
// [DebuggerDisplay("Person = FN {FirstName}, LN {LastName}")]
[DebuggerDisplay("Person = FN {"+nameof(Person.FirstName)+"}, LN
{"+nameof(Person.LastName)+"}")]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set;}
    // ...
}
```

Immutabilità delle stringhe

Le stringhe sono immutabili. Non puoi semplicemente cambiare la stringa esistente. Qualsiasi operazione sulla stringa crea una nuova istanza della stringa con un nuovo valore. Significa che se è necessario sostituire un singolo carattere in una stringa molto lunga, la memoria verrà allocata per un nuovo valore.

```
string veryLongString = ...
// memory is allocated
string newString = veryLongString.Remove(0,1); // removes first character of the string.
```

Se è necessario eseguire molte operazioni con valore stringa, utilizzare la classe `StringBuilder` che è progettata per una manipolazione efficiente delle stringhe:

```
var sb = new StringBuilder(someInitialString);
foreach(var str in manyManyStrings)
{
    sb.Append(str);
}
var finalString = sb.ToString();
```

Ompare le corde

Nonostante `String` sia un operatore di riferimento, l'operatore `==` confronta i valori stringa anziché i riferimenti.

Come forse saprai, la `string` è solo una serie di personaggi. Ma se pensi che il controllo e il

paragone delle stringhe siano fatti carattere per carattere, ti stai sbagliando. Questa operazione è specifica per la cultura (vedi Note sotto): alcune sequenze di caratteri possono essere considerate uguali a seconda della cultura .

Pensaci due volte prima di corteggiare il controllo di uguaglianza confrontando le proprietà di `Length` di due stringhe!

Utilizzare il sovraccarico del metodo `String.Equals` che accetta il valore di enumerazione `StringComparison` aggiuntivo, se è necessario modificare il comportamento predefinito.

Leggi stringhe online: https://riptutorial.com/it/dot-net/topic/2227/stringhe

Capitolo 47: System.Diagnostics

Examples

Cronometro

Questo esempio mostra come è possibile utilizzare il `Stopwatch` per confrontare un blocco di codice.

```
using System;
using System.Diagnostics;

public class Benchmark : IDisposable
{
    private Stopwatch sw;

    public Benchmark()
    {
        sw = Stopwatch.StartNew();
    }

    public void Dispose()
    {
        sw.Stop();
        Console.WriteLine(sw.Elapsed);
    }
}

public class Program
{
    public static void Main()
    {
        using (var bench = new Benchmark())
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

Esegui comandi shell

```
string strCmdText = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
System.Diagnostics.Process.Start("CMD.exe",strCmdText);
```

Questo per nascondere la finestra di cmd.

```
System.Diagnostics.Process process = new System.Diagnostics.Process();
System.Diagnostics.ProcessStartInfo startInfo = new System.Diagnostics.ProcessStartInfo();
startInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
startInfo.FileName = "cmd.exe";
startInfo.Arguments = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
process.StartInfo = startInfo;
process.Start();
```

Invia comando a CMD e Ricevi output

Questo metodo consente di inviare un `command` a `Cmd.exe` e restituisce l'output standard (incluso l'errore standard) come stringa:

```
private static string SendCommand(string command)
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
        WorkingDirectory = @"C:\Windows\System32", // Directory to make the call from
        WindowStyle = ProcessWindowStyle.Hidden,   // Hide the window
        UseShellExecute = false,                   // Do not use the OS shell to start the
process
        CreateNoWindow = true,                     // Start the process in a new window
        RedirectStandardOutput = true,             // This is required to get STDOUT
        RedirectStandardError = true               // This is required to get STDERR
    };

    var p = new Process {StartInfo = startInfo};

    p.Start();

    p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
    p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

uso

```
var servername = "SVR-01.domain.co.za";
var currentUsers = SendCommand($"/C QUERY USER /SERVER:{servername}")
```

Produzione

> string currentUsers = "USERNAME SESSIONNAME ID STATE IDLE TIME LOGON TIME Joe.Bloggs ica-cgp # 0 2 Attivo 24692 + 13: 29 25/07/2016 07:50 Jim.McFlannegan ica-cgp # 1 3 Attivo. 25/07 / 2016 08:33 Andy.McAnderson ica-cgp # 2 4 Attivo. 25/07/2016 08:54 John.Smith ica-cgp # 4 5 Attivo 14 25/07/2016 08:57 Bob.Bobbington ica-cgp # 5 6 Attivo 24692 + 13: 29 25/07/2016 09:05 Tim.Tom ica-cgp # 6 7 Attivo. 25/07/2016 09:08 Bob.Joges ica-cgp # 7 8 Attivo 24692 + 13: 29 25 / 07/2016 09:13 "

In alcune occasioni, l'accesso al server in questione potrebbe essere limitato a determinati utenti. Se si dispone delle credenziali di accesso per questo utente, è possibile inviare query con questo metodo:

```
private static string SendCommand(string command)
```

```
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
        WorkingDirectory = @"C:\Windows\System32",
        WindowStyle = ProcessWindowStyle.Hidden,    // This does not actually work in
conjunction with "runas" - the console window will still appear!
        UseShellExecute = false,
        CreateNoWindow = true,
        RedirectStandardOutput = true,
        RedirectStandardError = true,

        Verb = "runas",
        Domain = "doman1.co.za",
        UserName = "administrator",
        Password = GetPassword()
    };

    var p = new Process {StartInfo = startInfo};

    p.Start();

    p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
    p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

Ottenere la password:

```
static SecureString GetPassword()
{
    var plainText = "password123";
    var ss = new SecureString();
    foreach (char c in plainText)
    {
        ss.AppendChar(c);
    }

    return ss;
}
```

Gli appunti

Entrambi i metodi precedenti restituiranno una concatenazione di STDOUT e STDERR, poiché `OutputDataReceived` e `ErrorDataReceived` si `ErrorDataReceived` entrambi alla stessa variabile: `cmdOut` .

Leggi System.Diagnostics online: https://riptutorial.com/it/dot-net/topic/3143/system-diagnostics

Capitolo 48: System.IO

Examples

Leggere un file di testo usando StreamReader

```
string fullOrRelativePath = "testfile.txt";

string fileData;

using (var reader = new StreamReader(fullOrRelativePath))
{
    fileData = reader.ReadToEnd();
}
```

Si noti che questo sovraccarico del costruttore `StreamReader` esegue alcuni rilevamenti di codifica automatica, che possono o meno essere conformi alla codifica effettiva utilizzata nel file.

Si noti che esistono alcuni metodi di comodità che leggono tutto il testo dai file disponibili nella classe `System.IO.File` , ovvero `File.ReadAllText(path)` e `File.ReadAllLines(path)` .

Lettura / Scrittura di dati mediante System.IO.File

Innanzitutto, vediamo tre diversi modi di estrarre i dati da un file.

```
string fileText = File.ReadAllText(file);
string[] fileLines = File.ReadAllLines(file);
byte[] fileBytes = File.ReadAllBytes(file);
```

- Sulla prima riga, leggiamo tutti i dati nel file come una stringa.
- Sulla seconda riga, leggiamo i dati nel file in un array di stringhe. Ogni riga nel file diventa un elemento nell'array.
- Sul terzo leggiamo i byte dal file.

Successivamente, vediamo tre diversi metodi per **aggiungere** dati a un file. Se il file specificato non esiste, ciascun metodo creerà automaticamente il file prima di tentare di aggiungervi i dati.

```
 File.AppendAllText(file, "Here is some data that is\nappended to the file.");
 File.AppendAllLines(file, new string[2] { "Here is some data that is", "appended to the
file." });
 using (StreamWriter stream = File.AppendText(file))
 {
     stream.WriteLine("Here is some data that is");
     stream.Write("appended to the file.");
 }
```

- Sulla prima riga aggiungiamo semplicemente una stringa alla fine del file specificato.
- Sulla seconda riga aggiungiamo ogni elemento della matrice su una nuova riga nel file.

- Finalmente sulla terza riga usiamo `File.AppendText` per aprire uno streamwriter che aggiungerà qualsiasi dato vi sia scritto.

E infine, vediamo tre diversi metodi di **scrittura dei** dati in un file. La differenza tra l' *aggiunta* e la *scrittura* è che la scrittura **sovrascrive** i dati nel file mentre l'aggiunta **aggiunge** ai dati nel file. Se il file specificato non esiste, ciascun metodo creerà automaticamente il file prima di tentare di scrivere i dati su di esso.

```
File.WriteAllText(file, "here is some data\nin this file.");
File.WriteAllLines(file, new string[2] { "here is some data", "in this file" });
File.WriteAllBytes(file, new byte[2] { 0, 255 });
```

- La prima riga scrive una stringa nel file.
- La seconda riga scrive ogni stringa nell'array sulla propria riga nel file.
- E la terza riga consente di scrivere un array di byte nel file.

Porte seriali che utilizzano System.IO.SerialPorts

Iterazione su porte seriali connesse

```
using System.IO.Ports;
string[] ports = SerialPort.GetPortNames();
for (int i = 0; i < ports.Length; i++)
{
    Console.WriteLine(ports[i]);
}
```

Istanziazione di un oggetto System.IO.SerialPort

```
using System.IO.Ports;
SerialPort port = new SerialPort();
SerialPort port = new SerialPort("COM 1"); ;
SerialPort port = new SerialPort("COM 1", 9600);
```

NOTA : Questi sono solo tre dei sette overload del costruttore per il tipo SerialPort.

Lettura / Scrittura di dati su SerialPort

Il modo più semplice è utilizzare i metodi `SerialPort.Read` e `SerialPort.Write` . Tuttavia, è anche possibile recuperare un oggetto `System.IO.Stream` che è possibile utilizzare per lo streaming di dati su SerialPort. Per fare ciò, utilizzare `SerialPort.BaseStream` .

Lettura

```
int length = port.BytesToRead;
```

```
//Note that you can swap out a byte-array for a char-array if you prefer.
byte[] buffer = new byte[length];
port.Read(buffer, 0, length);
```

Puoi anche leggere tutti i dati disponibili:

```
string curData = port.ReadExisting();
```

O semplicemente leggere la prima nuova riga incontrata nei dati in arrivo:

```
string line = port.ReadLine();
```

scrittura

Il modo più semplice per scrivere dati su SerialPort è:

```
port.Write("here is some text to be sent over the serial port.");
```

Tuttavia puoi anche inviare i dati in questo modo quando necessario:

```
//Note that you can swap out the byte-array with a char-array if you so choose.
byte[] data = new byte[1] { 255 };
port.Write(data, 0, data.Length);
```

Leggi System.IO online: https://riptutorial.com/it/dot-net/topic/5259/system-io

Capitolo 49: System.IO.File class

Sintassi

- fonte di stringhe;
- destinazione stringa;

Parametri

Parametro	Dettagli
`source`	Il file che deve essere spostato in un'altra posizione.
`destination`	La directory in cui si desidera spostare l' `source` (questa variabile dovrebbe contenere anche il nome (e l'estensione del file) del file.

Examples

Elimina un file

Cancellare un file (se hai le autorizzazioni richieste) è semplice come:

```
File.Delete(path);
```

Tuttavia molte cose potrebbero andare storte:

- Non hai le autorizzazioni necessarie (viene lanciata `UnauthorizedAccessException`).
- Il file potrebbe essere in uso da qualcun altro (viene lanciata `IOException`).
- Il file non può essere eliminato a causa di un errore di basso livello o il supporto è di sola lettura (viene generata `IOException`).
- Il file non esiste più (viene lanciata `IOException`).

Nota che l'ultimo punto (il file non esiste) viene solitamente *aggirato* con uno snippet di codice come questo:

```
if (File.Exists(path))
    File.Delete(path);
```

Tuttavia non è un'operazione atomica e il file potrebbe essere eliminato da qualcun altro tra la chiamata a `File.Exists()` e prima di `File.Delete()` . L'approccio corretto per gestire le operazioni di I / O richiede la gestione delle eccezioni (presupponendo che una procedura alternativa possa essere intrapresa quando l'operazione fallisce):

```
if (File.Exists(path))
```

```
{
    try
    {
        File.Delete(path);
    }
    catch (IOException exception)
    {
        if (!File.Exists(path))
            return; // Someone else deleted this file

        // Something went wrong...
    }
    catch (UnauthorizedAccessException exception)
    {
        // I do not have required permissions
    }
}
```

Si noti che questi errori di I / O a volte sono transitori (file in uso, ad esempio) e se è coinvolta una connessione di rete, può automaticamente ripristinarsi senza alcuna azione da parte nostra. È quindi *normale riprovare* un'operazione di I / O alcune volte con un piccolo ritardo tra ogni tentativo:

```
public static void Delete(string path)
{
    if (!File.Exists(path))
        return;

    for (int i=1; ; ++i)
    {
        try
        {
            File.Delete(path);
            return;
        }
        catch (IOException e)
        {
            if (!File.Exists(path))
                return;

            if (i == NumberOfAttempts)
                throw;

            Thread.Sleep(DelayBetweenEachAttempt);
        }

        // You may handle UnauthorizedAccessException but this issue
        // will probably won't be fixed in few seconds...
    }
}

private const int NumberOfAttempts = 3;
private const int DelayBetweenEachAttempt = 1000; // ms
```

Nota: in Windows il file di ambiente non verrà realmente cancellato quando si chiama questa funzione, se qualcun altro apre il file utilizzando `FileShare.Delete` allora il file può essere cancellato ma in realtà si verifica solo quando il proprietario chiude il file.

Striscia le linee indesiderate da un file di testo

Cambiare un file di testo non è facile perché il suo contenuto deve essere spostato. Per i file di *piccole* dimensioni, il metodo più semplice è leggerne il contenuto in memoria e quindi riscrivere il testo modificato.

In questo esempio leggiamo tutte le linee da un file e rilasciamo tutte le righe vuote, quindi scriviamo di nuovo sul percorso originale:

```
File.WriteAllLines(path,
    File.ReadAllLines(path).Where(x => !String.IsNullOrWhiteSpace(x)));
```

Se il file è troppo grande per caricarlo in memoria e il percorso di output è diverso dal percorso di input:

```
File.WriteAllLines(outputPath,
    File.ReadLines(inputPath).Where(x => !String.IsNullOrWhiteSpace(x)));
```

Converti la codifica dei file di testo

Il testo viene salvato codificato (vedi anche argomento Stringhe), quindi a volte potrebbe essere necessario modificare la codifica, in questo esempio si presuppone (per semplicità) che il file non sia troppo grande e possa essere letto interamente in memoria:

```
public static void ConvertEncoding(string path, Encoding from, Encoding to)
{
    File.WriteAllText(path, File.ReadAllText(path, from), to);
}
```

Quando si eseguono le conversioni non dimenticare che il file può contenere BOM (Byte Order Mark), per capire meglio come viene gestito fare riferimento a Encoding.UTF8.GetString non tiene conto del Preamble / BOM .

"Tocca" una grande quantità di file (per aggiornare l'ultima ora di scrittura)

Questo esempio aggiorna l'ultimo tempo di scrittura di un numero enorme di file (utilizzando `System.IO.Directory.EnumerateFiles` anziché `System.IO.Directory.GetFiles()`). Opzionalmente è possibile specificare un modello di ricerca (il valore predefinito è `"*.*"` E alla fine cercare attraverso un albero di directory (non solo la directory specificata):

```
public static void Touch(string path,
                         string searchPattern = "*.*",
                         SearchOptions options = SearchOptions.None)
{
    var now = DateTime.Now;

    foreach (var filePath in Directory.EnumerateFiles(path, searchPattern, options))
    {
        File.SetLastWriteTime(filePath, now);
    }
```

```
}
```

Enumerare i file più vecchi di una quantità specificata

Questo snippet è una funzione di supporto per enumerare tutti i file più vecchi di un'età specificata, è utile, ad esempio, quando è necessario eliminare vecchi file di registro o vecchi dati memorizzati nella cache.

```
static IEnumerable<string> EnumerateAllFilesOlderThan(
                                TimeSpan maximumAge,
                                string path,
                                string searchPattern = "*.*",
                                SearchOption options = SearchOption.TopDirectoryOnly)
{
    DateTime oldestWriteTime = DateTime.Now - maximumAge;

    return Directory.EnumerateFiles(path, searchPattern, options)
        .Where(x => Directory.GetLastWriteTime(x) < oldestWriteTime);
}
```

Usato in questo modo:

```
var oldFiles = EnumerateAllFilesOlderThan(TimeSpan.FromDays(7), @"c:\log", "*.log");
```

Poche cose da notare:

- La ricerca viene eseguita utilizzando `Directory.EnumerateFiles()` anziché `Directory.GetFiles()` . L'enumerazione è *attiva,* quindi non è necessario attendere fino a quando tutte le voci del file system sono state recuperate.
- Stiamo controllando l'ora dell'ultima scrittura, ma è possibile utilizzare il tempo di creazione o l'ultimo orario di accesso (ad esempio per eliminare *i* file memorizzati in cache, si noti che il tempo di accesso potrebbe essere disabilitato).
- La granularità non è uniforme per tutte queste proprietà (tempo di scrittura, tempo di accesso, tempo di creazione), controllare MSDN per dettagli su questo.

Sposta un file da una posizione a un'altra

File.Move

Per spostare un file da una posizione a un'altra, una semplice riga di codice può ottenere ciò:

```
File.Move(@"C:\TemporaryFile.txt", @"C:\TemporaryFiles\TemporaryFile.txt");
```

Tuttavia, ci sono molte cose che potrebbero andare storte con questa semplice operazione. Ad esempio, cosa succede se l'utente che esegue il programma non ha un'unità con l'etichetta "C"? E se lo facessero, ma hanno deciso di rinominarlo in "B" o "M"?

Cosa succede se il file di origine (il file in cui desideri spostarti) è stato spostato a tua insaputa - o se semplicemente non esiste.

Questo può essere aggirato verificando prima se il file sorgente esiste:

```
string source = @"C:\TemporaryFile.txt", destination = @"C:\TemporaryFiles\TemporaryFile.txt";
if(File.Exists("C:\TemporaryFile.txt"))
{
    File.Move(source, destination);
}
```

Ciò assicurerà che in quel preciso istante il file esista e possa essere spostato in un'altra posizione. Ci possono essere momenti in cui una semplice chiamata a `File.Exists` non sarà sufficiente. In caso contrario, ricontrollare, comunicare all'utente che l'operazione non è riuscita o gestire l'eccezione.

Una eccezione `FileNotFoundException` non è l'unica che è probabile incontrare.

Vedi sotto per possibili eccezioni:

Tipo di eccezione	Descrizione
`IOException`	Il file esiste già o il file di origine non è stato trovato.
`ArgumentNullException`	Il valore dei parametri di origine e / o destinazione è nullo.
`ArgumentException`	Il valore dei parametri Source e / o Destination è vuoto o contiene caratteri non validi.
`UnauthorizedAccessException`	Non hai le autorizzazioni necessarie per eseguire questa azione.
`PathTooLongException`	L'origine, la destinazione o il percorso specificato superano la lunghezza massima. Su Windows, la lunghezza di un percorso deve essere inferiore a 248 caratteri, mentre i nomi dei file devono essere inferiori a 260 caratteri.
`DirectoryNotFoundException`	La directory specificata non è stata trovata.
`NotSupportedException`	I percorsi di origine o destinazione o i nomi dei file sono in un formato non valido.

Leggi System.IO.File class online: https://riptutorial.com/it/dot-net/topic/5395/system-io-file-class

Capitolo 50: System.Net.Mail

Osservazioni

È importante eliminare un System.Net.MailMessage perché ogni singolo allegato contiene un flusso e questi flussi devono essere liberati il prima possibile. L'istruzione using garantisce che l'oggetto usa e getta sia eliminato anche in caso di eccezioni

Examples

MailMessage

Ecco l'esempio della creazione di messaggi di posta con allegati. Dopo la creazione, inviamo questo messaggio con l'aiuto della classe `SmtpClient` . Qui viene utilizzata la porta predefinita 25.

```
public class clsMail
{
    private static bool SendMail(string mailfrom, List<string>replytos, List<string> mailtos,
List<string> mailccs, List<string> mailbccs, string body, string subject, List<string>
Attachment)
    {
        try
        {
            using(MailMessage MyMail = new MailMessage())
            {
                MyMail.From = new MailAddress(mailfrom);
                foreach (string mailto in mailtos)
                    MyMail.To.Add(mailto);

                if (replytos != null && replytos.Any())
                {
                    foreach (string replyto in replytos)
                        MyMail.ReplyToList.Add(replyto);
                }

                if (mailccs != null && mailccs.Any())
                {
                    foreach (string mailcc in mailccs)
                        MyMail.CC.Add(mailcc);
                }

                if (mailbccs != null && mailbccs.Any())
                {
                    foreach (string mailbcc in mailbccs)
                        MyMail.Bcc.Add(mailbcc);
                }

                MyMail.Subject = subject;
                MyMail.IsBodyHtml = true;
                MyMail.Body = body;
                MyMail.Priority = MailPriority.Normal;

                if (Attachment != null && Attachment.Any())
```

```
                {
                    System.Net.Mail.Attachment attachment;
                    foreach (var item in Attachment)
                    {
                        attachment = new System.Net.Mail.Attachment(item);
                        MyMail.Attachments.Add(attachment);
                    }
                }

                SmtpClient smtpMailObj = new SmtpClient();
                smtpMailObj.Host = "your host";
                smtpMailObj.Port = 25;
                smtpMailObj.Credentials = new System.Net.NetworkCredential("uid", "pwd");

                smtpMailObj.Send(MyMail);
                return true;
            }
        }
        catch
        {
            return false;
        }
    }
}
```

Mail con allegato

`MailMessage` rappresenta il messaggio di posta che può essere inviato ulteriormente utilizzando la classe `SmtpClient` . Diversi allegati (file) possono essere aggiunti al messaggio di posta.

```
using System.Net.Mail;

using(MailMessage myMail = new MailMessage())
{
     Attachment attachment = new Attachment(path);
     myMail.Attachments.Add(attachment);

     // further processing to send the mail message

}
```

Leggi System.Net.Mail online: https://riptutorial.com/it/dot-net/topic/7440/system-net-mail

Capitolo 51: System.Reflection.Emit spazio dei nomi

Examples

Creazione di un assieme in modo dinamico

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class DemoAssemblyBuilder
{
    public static void Main()
    {
        // An assembly consists of one or more modules, each of which
        // contains zero or more types. This code creates a single-module
        // assembly, the most common case. The module contains one type,
        // named "MyDynamicType", that has a private field, a property
        // that gets and sets the private field, constructors that
        // initialize the private field, and a method that multiplies
        // a user-supplied number by the private field value and returns
        // the result. In C# the type might look like this:
        /*
        public class MyDynamicType
        {
            private int m_number;

            public MyDynamicType() : this(42) {}
            public MyDynamicType(int initNumber)
            {
                m_number = initNumber;
            }

            public int Number
            {
                get { return m_number; }
                set { m_number = value; }
            }

            public int MyMethod(int multiplier)
            {
                return m_number * multiplier;
            }
        }
        */

        AssemblyName aName = new AssemblyName("DynamicAssemblyExample");
        AssemblyBuilder ab =
            AppDomain.CurrentDomain.DefineDynamicAssembly(
                aName,
                AssemblyBuilderAccess.RunAndSave);

        // For a single-module assembly, the module name is usually
        // the assembly name plus an extension.
```

```
        ModuleBuilder mb =
            ab.DefineDynamicModule(aName.Name, aName.Name + ".dll");

        TypeBuilder tb = mb.DefineType(
            "MyDynamicType",
             TypeAttributes.Public);

        // Add a private field of type int (Int32).
        FieldBuilder fbNumber = tb.DefineField(
            "m_number",
            typeof(int),
            FieldAttributes.Private);

        // Next, we make a simple sealed method.
        MethodBuilder mbMyMethod = tb.DefineMethod(
            "MyMethod",
            MethodAttributes.Public,
            typeof(int),
            new[] { typeof(int) });

        ILGenerator il = mbMyMethod.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this - always the first argument of any instance
method
        il.Emit(OpCodes.Ldfld, fbNumber);
        il.Emit(OpCodes.Ldarg_1); // Load the integer argument
        il.Emit(OpCodes.Mul); // Multiply the two numbers with no overflow checking
        il.Emit(OpCodes.Ret); // Return

        // Next, we build the property. This involves building the property itself, as well as
the
        // getter and setter methods.
        PropertyBuilder pbNumber = tb.DefineProperty(
            "Number", // Name
            PropertyAttributes.None,
            typeof(int), // Type of the property
            new Type[0]); // Types of indices, if any

        MethodBuilder mbSetNumber = tb.DefineMethod(
            "set_Number", // Name - setters are set_Property by convention
            // Setter is a special method and we don't want it to appear to callers from C#
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig |
MethodAttributes.Public | MethodAttributes.SpecialName,
            typeof(void), // Setters don't return a value
            new[] { typeof(int) }); // We have a single argument of type System.Int32

        // To generate the body of the method, we'll need an IL generator
        il = mbSetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldarg_1); // Load the new value
        il.Emit(OpCodes.Stfld, fbNumber); // Save the new value to this.m_number
        il.Emit(OpCodes.Ret); // Return

        // Finally, link the method to the setter of our property
        pbNumber.SetSetMethod(mbSetNumber);

        MethodBuilder mbGetNumber = tb.DefineMethod(
            "get_Number",
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig |
MethodAttributes.Public | MethodAttributes.SpecialName,
            typeof(int),
            new Type[0]);
```

```
        il = mbGetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldfld, fbNumber); // Load the value of this.m_number
        il.Emit(OpCodes.Ret); // Return the value

        pbNumber.SetGetMethod(mbGetNumber);

        // Finally, we add the two constructors.
        // Constructor needs to call the constructor of the parent class, or another
constructor in the same class
        ConstructorBuilder intConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new[] { typeof(int) });
        il = intConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Call, typeof(object).GetConstructor(new Type[0])); // call parent's
constructor
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldarg_1); // our int argument
        il.Emit(OpCodes.Stfld, fbNumber); // store argument in this.m_number
        il.Emit(OpCodes.Ret);

        var parameterlessConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new Type[0]);
        il = parameterlessConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldc_I4_S, (byte)42); // load 42 as an integer constant
        il.Emit(OpCodes.Call, intConstructor); // call this(42)
        il.Emit(OpCodes.Ret);

        // And make sure the type is created
        Type ourType = tb.CreateType();

        // The types from the assembly can be used directly using reflection, or we can save
the assembly to use as a reference
        object ourInstance = Activator.CreateInstance(ourType);
        Console.WriteLine(ourType.GetProperty("Number").GetValue(ourInstance)); // 42

        // Save the assembly for use elsewhere. This is very useful for debugging - you can
use e.g. ILSpy to look at the equivalent IL/C# code.
        ab.Save(@"DynamicAssemblyExample.dll");
        // Using newly created type
        var myDynamicType = tb.CreateType();
        var myDynamicTypeInstance = Activator.CreateInstance(myDynamicType);

        Console.WriteLine(myDynamicTypeInstance.GetType()); // MyDynamicType

        var numberField = myDynamicType.GetField("m_number", BindingFlags.NonPublic |
BindingFlags.Instance);
        numberField.SetValue (myDynamicTypeInstance, 10);

        Console.WriteLine(numberField.GetValue(myDynamicTypeInstance)); // 10
    }
}
```

Leggi System.Reflection.Emit spazio dei nomi online: https://riptutorial.com/it/dot-net/topic/74/system-reflection-emit-spazio-dei-nomi

Capitolo 52: System.Runtime.Caching.MemoryCache (ObjectCache)

Examples

Aggiungere elementi alla cache (set)

La funzione set inserisce una voce cache nella cache utilizzando un'istanza CacheItem per fornire la chiave e il valore per la voce cache.

Questa funzione `ObjectCache.Set(CacheItem, CacheItemPolicy)` override di `ObjectCache.Set(CacheItem, CacheItemPolicy)`

```
private static bool SetToCache()
{
    string key = "Cache_Key";
    string value = "Cache_Value";

    //Get a reference to the default MemoryCache instance.
    var cacheContainer = MemoryCache.Default;

    var policy = new CacheItemPolicy()
    {
        AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(DEFAULT_CACHE_EXPIRATION_MINUTES)
     };
     var itemToCache = new CacheItem(key, value); //Value is of type object.
     cacheContainer.Set(itemToCache, policy);
}
```

System.Runtime.Caching.MemoryCache (ObjectCache)

Questa funzione ottiene la cache del modulo degli elementi esistente e, se l'elemento non esiste nella cache, recupererà l'elemento in base alla funzione valueFetchFactory.

```
    public static TValue GetExistingOrAdd<TValue>(string key, double minutesForExpiration,
Func<TValue> valueFetchFactory)
    {
        try
        {
            //The Lazy class provides Lazy initialization which will evaluate
            //the valueFetchFactory only if item is not in the cache.
            var newValue = new Lazy<TValue>(valueFetchFactory);

            //Setup the cache policy if item will be saved back to cache.
            CacheItemPolicy policy = new CacheItemPolicy()
            {
                AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(minutesForExpiration)
            };
```

```
            //returns existing item form cache or add the new value if it does not exist.
            var cachedItem = _cacheContainer.AddOrGetExisting(key, newValue, policy) as
Lazy<TValue>;

            return (cachedItem ?? newValue).Value;
        }
        catch (Exception excep)
        {
            return default(TValue);
        }
    }
```

Leggi System.Runtime.Caching.MemoryCache (ObjectCache) online: https://riptutorial.com/it/dot-net/topic/76/system-runtime-caching-memorycache--objectcache-

Capitolo 53: Task Parallel Library (TPL)

Osservazioni

Scopo e casi d'uso

Lo scopo della Task Parallel Library è di semplificare il processo di scrittura e mantenimento del codice multithreaded e parallelo.

Alcuni casi d'uso *:

- Mantenere un'interfaccia utente reattiva eseguendo il lavoro in background su un'attività separata
- Distribuzione del carico di lavoro
- Consentire a un'applicazione client di inviare e ricevere richieste contemporaneamente (resto, TCP / UDP, ect)
- Lettura e / o scrittura di più file contemporaneamente

* Il codice deve essere considerato caso per caso per il multithreading. Ad esempio, se un ciclo ha solo poche iterazioni o esegue solo una piccola parte del lavoro, il sovraccarico per il parallelismo può superare i benefici.

TPL con .Net 3.5

Il TPL è anche disponibile per .Net 3.5 incluso in un pacchetto NuGet, si chiama Task Parallel Library.

Examples

Ciclo base produttore-consumatore (BlockingCollection)

```
var collection = new BlockingCollection<int>(5);
var random = new Random();

var producerTask = Task.Run(() => {
    for(int item=1; item<=10; item++)
    {
        collection.Add(item);
        Console.WriteLine("Produced: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
    collection.CompleteAdding();
    Console.WriteLine("Producer completed!");
});
```

Vale la pena notare che se non si chiama `collection.CompleteAdding();` , puoi continuare ad aggiungere alla raccolta anche se l'attività dell'utente è in esecuzione. Basta chiamare

`collection.CompleteAdding();` quando sei sicuro che non ci siano più aggiunte. Questa funzionalità può essere utilizzata per creare un produttore multiplo su un modello consumer singolo in cui sono presenti più origini che alimentano elementi in BlockingCollection e un singolo utente che estrae elementi e ne fa qualcosa. Se BlockingCollection è vuoto prima di chiamare l'aggiunta completa, l'Enumerable from `collection.GetConsumingEnumerable()` bloccherà fino a quando un nuovo elemento non verrà aggiunto alla raccolta o BlockingCollection.CompleteAdding (); viene chiamato e la coda è vuota.

```
var consumerTask = Task.Run(() => {
    foreach(var item in collection.GetConsumingEnumerable())
    {
        Console.WriteLine("Consumed: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
    Console.WriteLine("Consumer completed!");
});

Task.WaitAll(producerTask, consumerTask);

Console.WriteLine("Everything completed!");
```

Compito: istanziazione di base e attesa

Un'attività può essere creata istanziando direttamente la classe `Task` ...

```
var task = new Task(() =>
{
    Console.WriteLine("Task code starting...");
    Thread.Sleep(2000);
    Console.WriteLine("...task code ending!");
});

Console.WriteLine("Starting task...");
task.Start();
task.Wait();
Console.WriteLine("Task completed!");
```

... o usando il metodo `Task.Run` statico:

```
Console.WriteLine("Starting task...");
var task = Task.Run(() =>
{
    Console.WriteLine("Task code starting...");
    Thread.Sleep(2000);
    Console.WriteLine("...task code ending!");
});
task.Wait();
Console.WriteLine("Task completed!");
```

Si noti che solo nel primo caso è necessario richiamare esplicitamente `Start` .

Attività: WaitAll e acquisizione variabile

```
var tasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() =>
{
    Console.WriteLine("I'm task " + n);
    return n;
})).ToArray();

foreach(var task in tasks) task.Start();
Task.WaitAll(tasks);

foreach(var task in tasks)
    Console.WriteLine(task.Result);
```

Compito: WaitAny

```
var allTasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() => n)).ToArray();
var pendingTasks = allTasks.ToArray();

foreach(var task in allTasks) task.Start();

while(pendingTasks.Length > 0)
{
    var finishedTask = pendingTasks[Task.WaitAny(pendingTasks)];
    Console.WriteLine("Task {0} finished", finishedTask.Result);
    pendingTasks = pendingTasks.Except(new[] {finishedTask}).ToArray();
}

Task.WaitAll(allTasks);
```

Nota: l' `WaitAll` finale è necessario perché `WaitAny` non fa in modo che vengano osservate eccezioni.

Attività: gestione delle eccezioni (utilizzando Attendi)

```
var task1 = Task.Run(() =>
{
    Console.WriteLine("Task 1 code starting...");
    throw new Exception("Oh no, exception from task 1!!");
});

var task2 = Task.Run(() =>
{
    Console.WriteLine("Task 2 code starting...");
    throw new Exception("Oh no, exception from task 2!!");
});

Console.WriteLine("Starting tasks...");
try
{
    Task.WaitAll(task1, task2);
}
catch(AggregateException ex)
{
    Console.WriteLine("Task(s) failed!");
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine(inner.Message);
}
```

```
Console.WriteLine("Task 1 status is: " + task1.Status); //Faulted
Console.WriteLine("Task 2 status is: " + task2.Status); //Faulted
```

Attività: gestione delle eccezioni (senza usare Wait)

```
var task1 = Task.Run(() =>
{
    Console.WriteLine("Task 1 code starting...");
    throw new Exception("Oh no, exception from task 1!!");
});

var task2 = Task.Run(() =>
{
    Console.WriteLine("Task 2 code starting...");
    throw new Exception("Oh no, exception from task 2!!");
});

var tasks = new[] {task1, task2};

Console.WriteLine("Starting tasks...");
while(tasks.All(task => !task.IsCompleted));

foreach(var task in tasks)
{
    if(task.IsFaulted)
        Console.WriteLine("Task failed: " +
            task.Exception.InnerExceptions.First().Message);
}

Console.WriteLine("Task 1 status is: " + task1.Status); //Faulted
Console.WriteLine("Task 2 status is: " + task2.Status); //Faulted
```

Compito: cancellare usando CancellationToken

```
var cancellationTokenSource = new CancellationTokenSource();
var cancellationToken = cancellationTokenSource.Token;

var task = new Task((state) =>
    {
        int i = 1;
        var myCancellationToken = (CancellationToken)state;
        while(true)
        {
            Console.Write("{0} ", i++);
            Thread.Sleep(1000);
            myCancellationToken.ThrowIfCancellationRequested();
        }
    },
    cancellationToken: cancellationToken,
    state: cancellationToken);

Console.WriteLine("Counting to infinity. Press any key to cancel!");
task.Start();
Console.ReadKey();

cancellationTokenSource.Cancel();
try
```

```
{
    task.Wait();
}
catch(AggregateException ex)
{
    ex.Handle(inner => inner is OperationCanceledException);
}

Console.WriteLine($"{Environment.NewLine}You have cancelled! Task status is: {task.Status}");
//Canceled
```

In alternativa a `ThrowIfCancellationRequested` , la richiesta di annullamento può essere rilevata con `IsCancellationRequested` e una `OperationCanceledException` può essere generata manualmente:

```
//New task delegate
int i = 1;
var myCancellationToken = (CancellationToken)state;
while(!myCancellationToken.IsCancellationRequested)
{
    Console.Write("{0} ", i++);
    Thread.Sleep(1000);
}
Console.WriteLine($"{Environment.NewLine}Ouch, I have been cancelled!!");
throw new OperationCanceledException(myCancellationToken);
```

Nota come il token di cancellazione viene passato al costruttore di task nel parametro `cancellationToken` . Ciò è necessario in modo che le transizioni compito alle `Canceled` stato, non alla `Faulted` stato, quando `ThrowIfCancellationRequested` viene richiamato. Inoltre, per lo stesso motivo, il token di cancellazione viene esplicitamente fornito nel costruttore di `OperationCanceledException` nel secondo caso.

Task.WhenAny

```
var random = new Random();
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(async() =>
{
    Console.WriteLine("I'm task " + n);
    await Task.Delay(random.Next(10,1000));
    return n;
}));

Task<Task<int>> whenAnyTask = Task.WhenAny(tasks);
Task<int> completedTask = await whenAnyTask;
Console.WriteLine("The winner is: task " + await completedTask);

await Task.WhenAll(tasks);
Console.WriteLine("All tasks finished!");
```

Task.WhenAll

```
var random = new Random();
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(() =>
{
    Console.WriteLine("I'm task " + n);
```

```
    return n;
}));

Task<int[]> task = Task.WhenAll(tasks);
int[] results = await task;

Console.WriteLine(string.Join(",", results.Select(n => n.ToString())));
// Output: 1,2,3,4,5
```

Parallel.Invoke

```
var actions = Enumerable.Range(1, 10).Select(n => new Action(() =>
{
    Console.WriteLine("I'm task " + n);
    if((n & 1) == 0)
        throw new Exception("Exception from task " + n);
})).ToArray();

try
{
    Parallel.Invoke(actions);
}
catch(AggregateException ex)
{
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine("Task failed: " + inner.Message);
}
```

Parallel.ForEach

Questo esempio utilizza `Parallel.ForEach` per calcolare la somma dei numeri tra 1 e 10000 utilizzando più thread. Per raggiungere la sicurezza del thread, `Interlocked.Add` viene utilizzato per sommare i numeri.

```
using System.Threading;

int Foo()
{
    int total = 0;
    var numbers = Enumerable.Range(1, 10000).ToList();
    Parallel.ForEach(numbers,
        () => 0, // initial value,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

Parallel.For

Questo esempio utilizza `Parallel.For` calcolare la somma dei numeri tra 1 e 10000 utilizzando più thread. Per raggiungere la sicurezza del thread, `Interlocked.Add` viene utilizzato per sommare i numeri.

```
using System.Threading;
```

```
int Foo()
{
    int total = 0;
    Parallel.For(1, 10001,
        () => 0, // initial value,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

Contesto di esecuzione fluente con AsyncLocal

Quando è necessario passare alcuni dati dall'attività padre alle attività figli, in modo tale da scorrere logicamente con l'esecuzione, utilizzare la classe `AsyncLocal` :

```
void Main()
{
    AsyncLocal<string> user = new AsyncLocal<string>();
    user.Value = "initial user";

    // this does not affect other tasks - values are local relative to the branches of
execution flow
    Task.Run(() => user.Value = "user from another task");

    var task1 = Task.Run(() =>
    {
        Console.WriteLine(user.Value); // outputs "initial user"
        Task.Run(() =>
        {
            // outputs "initial user" - value has flown from main method to this task without
being changed
            Console.WriteLine(user.Value);
        }).Wait();

        user.Value = "user from task1";

        Task.Run(() =>
        {
            // outputs "user from task1" - value has flown from main method to task1
            // than value was changed and flown to this task.
            Console.WriteLine(user.Value);
        }).Wait();
    });

    task1.Wait();

    // ouputs "initial user" - changes do not propagate back upstream the execution flow
    Console.WriteLine(user.Value);
}
```

Nota: Come si può vedere dall'esempio sopra, `AsynLocal.Value` ha una `copy on read` semantica, ma se si scorre qualche tipo di riferimento e si cambiano le sue proprietà si avranno effetti su altre attività. Pertanto, la migliore pratica con `AsyncLocal` consiste nell'utilizzare tipi di valore o tipi immutabili.

Parallel.ForEach in VB.NET

```
For Each row As DataRow In FooDataTable.Rows
    Me.RowsToProcess.Add(row)
Next

Dim myOptions As ParallelOptions = New ParallelOptions()
myOptions.MaxDegreeOfParallelism = environment.processorcount

Parallel.ForEach(RowsToProcess, myOptions, Sub(currentRow, state)
                                               ProcessRowParallel(currentRow, state)
                                           End Sub)
```

Attività: restituire un valore

L'attività che restituisce un valore restituisce il tipo di `Task< TResult >` dove TResult è il tipo di valore che deve essere restituito. È possibile interrogare il risultato di un'attività tramite la sua proprietà Result.

```
Task<int> t = Task.Run(() =>
    {
        int sum = 0;

        for(int i = 0; i < 500; i++)
            sum += i;

        return sum;
    });

Console.WriteLine(t.Result); // Outuput 124750
```

Se l'attività viene eseguita in modo asincrono rispetto all'attesa, l'operazione restituisce il risultato.

```
public async Task DoSomeWork()
{
    WebClient client = new WebClient();
    // Because the task is awaited, result of the task is assigned to response
    string response = await client.DownloadStringTaskAsync("http://somedomain.com");
}
```

Leggi Task Parallel Library (TPL) online: https://riptutorial.com/it/dot-net/topic/55/task-parallel-library--tpl-

Capitolo 54: Test unitario

Examples

Aggiunta del progetto di test dell'unità MSTest a una soluzione esistente

- Fare clic con il tasto destro sulla soluzione, Aggiungi nuovo progetto
- Dalla sezione Test, selezionare un Progetto Test unità
- Scegli un nome per il montaggio: se stai collaudando il progetto `Foo` , il nome può essere `Foo.Tests`
- Aggiungi un riferimento al progetto testato nei riferimenti del progetto di test unitario

Creazione di un metodo di prova di esempio

MSTest (il framework di testing predefinito) richiede che le classi di test `[TestClass]` decorate da un attributo `[TestClass]` e i metodi di test con un attributo `[TestMethod]` e siano pubblici.

```
[TestClass]
public class FizzBuzzFixture
{
    [TestMethod]
    public void Test1()
    {
        //arrange
        var solver = new FizzBuzzSolver();
        //act
        var result = solver.FizzBuzz(1);
        //assert
        Assert.AreEqual("1",result);
    }
}
```

Leggi Test unitario online: https://riptutorial.com/it/dot-net/topic/5365/test-unitario

Capitolo 55: threading

Examples

Accesso ai controlli del modulo da altri thread

Se si desidera modificare un attributo di un controllo come una casella di testo o un'etichetta da un altro thread rispetto al thread della GUI che ha creato il controllo, sarà necessario richiamarlo altrimenti si potrebbe ottenere un messaggio di errore che indica:

> "Operazione cross-thread non valida: controllo 'control_name' accessibile da un thread diverso dal thread su cui è stato creato."

L'uso di questo codice di esempio su un modulo system.windows.forms genera un'eccezione con quel messaggio:

```
private void button4_Click(object sender, EventArgs e)
{
    Thread thread = new Thread(updatetextbox);
    thread.Start();
}

private void updatetextbox()
{
    textBox1.Text = "updated"; // Throws exception
}
```

Invece quando si desidera modificare il testo di una casella di testo all'interno di un thread che non lo possiede, utilizzare Control.Invoke o Control.BeginInvoke. È inoltre possibile utilizzare Control.InvokeRequired per verificare se è necessario il richiamo del controllo.

```
private void updatetextbox()
{
    if (textBox1.InvokeRequired)
        textBox1.BeginInvoke((Action)(() => textBox1.Text = "updated"));
    else
        textBox1.Text = "updated";
}
```

Se è necessario farlo spesso, è possibile scrivere un'estensione per oggetti richiamabili per ridurre la quantità di codice necessaria per effettuare questo controllo:

```
public static class Extensions
{
    public static void BeginInvokeIfRequired(this ISynchronizeInvoke obj, Action action)
    {
        if (obj.InvokeRequired)
            obj.BeginInvoke(action, new object[0]);
        else
            action();
    }
```

```
}
```

E l'aggiornamento della casella di testo da qualsiasi thread diventa un po 'più semplice:

```
private void updatetextbox()
{
    textBox1.BeginInvokeIfRequired(() => textBox1.Text = "updated");
}
```

Tenere presente che Control.BeginInvoke come in questo esempio è asincrono, il che significa che il codice proveniente dopo una chiamata a Control.BeginInvoke può essere eseguito immediatamente dopo, indipendentemente dal fatto che il delegato passato sia stato eseguito o meno.

Se devi essere sicuro che textBox1 sia aggiornato prima di continuare, usa invece Control.Invoke, che bloccherà il thread chiamante fino a quando il tuo delegato non sarà stato eseguito. Si noti che questo approccio può rallentare notevolmente il codice se si effettuano molte chiamate invocate e si noti che l'applicazione si bloccherà se il thread della GUI è in attesa che il thread chiamante completi o rilasci una risorsa trattenuta.

Leggi threading online: https://riptutorial.com/it/dot-net/topic/3098/threading

Capitolo 56: Tipi personalizzati

Osservazioni

In genere una `struct` viene utilizzata solo quando le prestazioni sono molto importanti. Poiché i tipi di valore vivono nello stack, è possibile accedervi molto più rapidamente delle classi. Tuttavia, lo stack ha molto meno spazio dell'heap, quindi le strutture dovrebbero essere ridotte (Microsoft consiglia `struct` s occupare non più di 16 byte).

Una `class` è il tipo più usato (di questi tre) in C #, ed è generalmente quello che dovresti fare prima.

Un `enum` viene utilizzato ogni volta che è possibile avere un elenco chiaramente definito di elementi che devono essere definiti solo una volta (in fase di compilazione). Le enumerazioni sono utili ai programmatori come riferimento leggero ad alcuni valori: invece di definire un elenco di variabili `constant` da confrontare, puoi usare un enum e ottenere il supporto Intellisense per assicurarti di non utilizzare accidentalmente un valore errato.

Examples

Definizione di Struct

Le strutture ereditano da System.ValueType, sono tipi di valore e vivono nello stack. Quando i tipi di valore vengono passati come parametro, vengono passati per valore.

```
Struct MyStruct
{
    public int x;
    public int y;
}
```

Passato per valore significa che il valore del parametro viene *copiato* per il metodo e qualsiasi modifica apportata al parametro nel metodo non viene riflessa all'esterno del metodo. Ad esempio, si consideri il seguente codice, che chiama un metodo denominato `AddNumbers` , passando le variabili `a` e `b` , che sono di tipo `int` , che è un tipo di valore.

```
int a = 5;
int b = 6;

AddNumbers(a,b);

public AddNumbers(int x, int y)
{
    int z = x + y; // z becomes 11
    x = x + 5; // now we changed x to be 10
```

```
    z = x + y; // now z becomes 16
}
```

Anche se abbiamo aggiunto da 5 a `x` all'interno del metodo, il valore di `a` rimane invariato, perché è un tipo Value, e ciò significa che `x` era una *copia* del valore di `a` valore, ma non in realtà `a` .

Ricorda, i tipi di valore vivono nello stack e vengono passati per valore.

Definizione di classe

Le classi ereditano da System.Object, sono tipi di riferimento e vivono nell'heap. Quando i tipi di riferimento vengono passati come parametro, vengono passati per riferimento.

```
public Class MyClass
{
    public int a;
    public int b;
}
```

Passato per riferimento significa che un *riferimento* al parametro viene passato al metodo e tutte le modifiche al parametro si rifletteranno al di fuori del metodo quando ritorna, poiché il riferimento è *esattamente allo stesso oggetto in memoria* . Usiamo lo stesso esempio di prima, ma prima "intaschiamo" l' `int` s in una classe.

```
MyClass instanceOfMyClass = new MyClass();
instanceOfMyClass.a = 5;
instanceOfMyClass.b = 6;

AddNumbers(instanceOfMyClass);

public AddNumbers(MyClass sample)
{
    int z = sample.a + sample.b; // z becomes 11
    sample.a = sample.a + 5; // now we changed a to be 10
    z = sample.a + sample.b; // now z becomes 16
}
```

Questa volta, quando abbiamo cambiato `sample.a` a `10` , cambia *anche* il valore di `instanceOfMyClass.a` , perché è stato *passato per riferimento* . Passato per riferimento significa che un *riferimento* (anche a volte chiamato *puntatore*) all'oggetto è stato passato al metodo, anziché una copia dell'oggetto stesso.

Ricorda, i tipi di riferimento vivono nell'heap e vengono passati per riferimento.

Definizione Enum

Un enum è un tipo speciale di classe. La parola chiave `enum` dice al compilatore che questa classe eredita dalla classe System.Enum astratta. Le enumerazioni vengono utilizzate per elenchi distinti di elementi.

```
public enum MyEnum
{
    Monday = 1,
    Tuesday,
    Wednesday,
    //...
}
```

Puoi pensare a un enum come un modo conveniente di mappare le costanti ad un valore sottostante. L'enum definito sopra dichiara i valori per ogni giorno della settimana e inizia con `1` . `Tuesday` verrà automaticamente mappato a `2` , `Wednesday 3` , ecc.

Per impostazione predefinita, enum usa `int` come tipo sottostante e inizia da 0, ma puoi usare uno qualsiasi dei seguenti *tipi interi* : `byte, sbyte, short, ushort, int, uint, long, or ulong` e puoi specificare valori espliciti per qualsiasi articolo. Se alcuni elementi sono specificati in modo esplicito, ma alcuni non lo sono, ogni elemento dopo l'ultimo definito verrà incrementato di 1.

Vorremmo utilizzare questo esempio *lanciando* qualche altro valore a un *MyEnum* in questo modo:

```
MyEnum instance = (MyEnum)3; // the variable named 'instance' gets a
                             //value of MyEnum.Wednesday, which maps to 3.

int x = 2;
instance = (MyEnum)x; // now 'instance' has a value of MyEnum.Tuesday
```

Un altro utile, anche se più complesso, tipo di enum è chiamato `Flags` . *Decorando* un enum con le `Flags` di attributo, è possibile assegnare un più di un valore variabile alla volta. Si noti che quando si esegue questa operazione è *necessario* definire esplicitamente i valori nella rappresentazione di base 2.

```
[Flags]
public enum MyEnum
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 4,
    Thursday = 8,
    Friday = 16,
    Saturday = 32,
    Sunday = 64
}
```

Ora è possibile confrontare più di un valore alla volta, utilizzando *confronti bit a bit* o, se si utilizza

.NET 4.0 o `Enum.HasFlag` successive, il metodo `Enum.HasFlag` .

```
MyEnum instance = MyEnum.Monday | MyEnum.Thursday; // instance now has a value of
                                                   // *both* Monday and Thursday,
                                                   // represented by (in binary) 0100.

if (instance.HasFlag(MyEnum.Wednesday))
{
    // it doesn't, so this block is skipped
}
else if (instance.HasFlag(MyEnum.Thursday))
{
    // it does, so this block is executed
}
```

Poiché la classe Enum è sottoclasa da `System.ValueType` , viene trattata come un tipo di valore e passata per valore, non per riferimento. L'oggetto base viene creato sull'heap, ma quando si passa un valore enum in una chiamata di funzione, una copia del valore utilizzando il tipo di valore sottostante dell'Enum (in genere System.Int32) viene inserita nello stack. Il compilatore tiene traccia dell'associazione tra questo valore e l'oggetto di base creato nello stack. Vedi ValueType Class (System) (MSDN) per ulteriori informazioni.

Leggi Tipi personalizzati online: https://riptutorial.com/it/dot-net/topic/57/tipi-personalizzati

Capitolo 57: TPL Dataflow

Osservazioni

Librerie usate negli esempi

`System.Threading.Tasks.Dataflow`

`System.Threading.Tasks`

`System.Net.Http`

`System.Net`

Differenza tra Post e SendAsync

Per aggiungere elementi a un blocco puoi utilizzare `Post` o `SendAsync` .

`Post` proverà ad aggiungere l'elemento in modo sincrono e restituirà un `bool` dice se è riuscito o meno. Potrebbe non riuscire quando, ad esempio, un blocco ha raggiunto la sua `BoundedCapcity` e non ha più spazio per nuovi elementi ancora. `SendAsync` altra parte `SendAsync` restituirà un `Task<bool>` che è possibile `await` . Tale attività verrà completata in futuro con un risultato `true` quando il blocco ha cancellato la coda interna e può accettare più articoli o con un risultato `false` se declina in modo permanente (ad es. A seguito di annullamento).

Examples

Pubblicazione su ActionBlock e in attesa di completamento

```
// Create a block with an asynchronous action
var block = new ActionBlock<string>(async hostName =>
{
    IPAddress[] ipAddresses = await Dns.GetHostAddressesAsync(hostName);
    Console.WriteLine(ipAddresses[0]);
});

block.Post("google.com"); // Post items to the block's InputQueue for processing
block.Post("reddit.com");
block.Post("stackoverflow.com");

block.Complete(); // Tell the block to complete and stop accepting new items
await block.Completion; // Asynchronously wait until all items completed processingu
```

Collegamento di blocchi per creare una pipeline

```
var httpClient = new HttpClient();
```

```
// Create a block the accepts a uri and returns its contents as a string
var downloaderBlock = new TransformBlock<string, string>(
    async uri => await httpClient.GetStringAsync(uri));

// Create a block that accepts the content and prints it to the console
var printerBlock = new ActionBlock<string>(
    contents => Console.WriteLine(contents));

// Make the downloaderBlock complete the printerBlock when its completed.
var dataflowLinkOptions = new DataflowLinkOptions {PropagateCompletion = true};

// Link the block to create a pipeline
downloaderBlock.LinkTo(printerBlock, dataflowLinkOptions);

// Post urls to the first block which will pass their contents to the second one.
downloaderBlock.Post("http://youtube.com");
downloaderBlock.Post("http://github.com");
downloaderBlock.Post("http://twitter.com");

downloaderBlock.Complete(); // Completion will propagate to printerBlock
await printerBlock.Completion; // Only need to wait for the last block in the pipeline
```

Produttore / consumatore sincrono con BufferBlock

```
public class Producer
{
    private static Random random = new Random((int)DateTime.UtcNow.Ticks);
    //produce the value that will be posted to buffer block
    public double Produce ( )
    {
        var value = random.NextDouble();
        Console.WriteLine($"Producing value: {value}");
        return value;
    }
}

public class Consumer
{
    //consume the value that will be received from buffer block
    public void Consume (double value) => Console.WriteLine($"Consuming value: {value}");
}

class Program
{
    private static BufferBlock<double> buffer = new BufferBlock<double>();
    static void Main (string[] args)
    {
        //start a task that will every 1 second post a value from the producer to buffer block
        var producerTask = Task.Run(async () =>
        {
            var producer = new Producer();
            while(true)
            {
                buffer.Post(producer.Produce());
                await Task.Delay(1000);
            }
        });
        //start a task that will recieve values from bufferblock and consume it
        var consumerTask = Task.Run(() =>
```

```
        {
            var consumer = new Consumer();
            while(true)
            {
                consumer.Consume(buffer.Receive());
            }
        });

        Task.WaitAll(new[] { producerTask, consumerTask });
    }
}
```

Consumatore di produttori asincroni con un buffer buffer limitato

```
var bufferBlock = new BufferBlock<int>(new DataflowBlockOptions
{
    BoundedCapacity = 1000
});

var cancellationToken = new CancellationTokenSource(TimeSpan.FromSeconds(10)).Token;

var producerTask = Task.Run(async () =>
{
    var random = new Random();

    while (!cancellationToken.IsCancellationRequested)
    {
        var value = random.Next();
        await bufferBlock.SendAsync(value, cancellationToken);
    }
});

var consumerTask = Task.Run(async () =>
{
    while (await bufferBlock.OutputAvailableAsync())
    {
        var value = bufferBlock.Receive();
        Console.WriteLine(value);
    }
});

await Task.WhenAll(producerTask, consumerTask);
```

Leggi TPL Dataflow online: https://riptutorial.com/it/dot-net/topic/784/tpl-dataflow

Capitolo 58: Usando il progresso e IProgress

Examples

Semplice resoconto dei progressi

`IProgress<T>` può essere utilizzato per segnalare lo stato di avanzamento di alcune procedure a un'altra procedura. Questo esempio mostra come è possibile creare un metodo di base che ne segnali i progressi.

```
void Main()
{
    IProgress<int> p = new Progress<int>(progress =>
    {
        Console.WriteLine("Running Step: {0}", progress);
    });
    LongJob(p);
}

public void LongJob(IProgress<int> progress)
{
    var max = 10;
    for (int i = 0; i < max; i++)
    {
        progress.Report(i);
    }
}
```

Produzione:

```
Running Step: 0
Running Step: 3
Running Step: 4
Running Step: 5
Running Step: 6
Running Step: 7
Running Step: 8
Running Step: 9
Running Step: 2
Running Step: 1
```

Si noti che quando si esegue questo codice, è possibile che i numeri vengano riprodotti fuori ordine. Questo perché il `IProgress<T>.Report()` viene eseguito in modo asincrono e pertanto non è adatto per le situazioni in cui è necessario riportare l'avanzamento in ordine.

Utilizzando IProgress

È importante notare che la `System.Progress<T>` non ha il metodo `Report()` disponibile su di esso. Questo metodo è stato implementato esplicitamente `IProgress<T>` e, pertanto, deve essere richiamato su `Progress<T>` quando viene `IProgress<T>` il cast su un `IProgress<T>` .

```
var p1 = new Progress<int>();
p1.Report(1); //compiler error, Progress does not contain method 'Report'

IProgress<int> p2 = new Progress<int>();
p2.Report(2); //works

var p3 = new Progress<int>();
((IProgress<int>)p3).Report(3); //works
```

Leggi Usando il progresso e IProgress online: https://riptutorial.com/it/dot-net/topic/5628/usando-il-progresso--t--e-iprogress--t-

Capitolo 59: XmlSerializer

Osservazioni

Non utilizzare `XmlSerializer` per analizzare `HTML` . Per questo, sono disponibili librerie speciali come l' HTML Agility Pack

Examples

Serializza oggetto

```
public void SerializeFoo(string fileName, Foo foo)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.Open(fileName, FileMode.Create))
    {
        serializer.Serialize(stream, foo);
    }
}
```

Deserializzare l'oggetto

```
public Foo DeserializeFoo(string fileName)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.OpenRead(fileName))
    {
        return (Foo)serializer.Deserialize(stream);
    }
}
```

Comportamento: mappa il nome dell'elemento in Proprietà

```
<Foo>
    <Dog/>
</Foo>
```

-

```
public class Foo
{
    // Using XmlElement
    [XmlElement(Name="Dog")]
    public Animal Cat { get; set; }
}
```

Comportamento: Mappare il nome dell'array sulla proprietà (XmlArray)

```
<Store>
    <Articles>
        <Product/>
        <Product/>
    </Articles>
</Store>
```

-

```
public class Store
{
    [XmlArray("Articles")]
    public List<Product> Products {get; set; }
}
```

Formattazione: formato DateTime personalizzato

```
public class Dog
{
    private const string _birthStringFormat = "yyyy-MM-dd";

    [XmlIgnore]
    public DateTime Birth {get; set;}

    [XmlElement(ElementName="Birth")]
    public string BirthString
    {
        get { return Birth.ToString(_birthStringFormat); }
        set { Birth = DateTime.ParseExact(value, _birthStringFormat,
CultureInfo.InvariantCulture); }
    }
}
```

Creazione efficiente di serializzatori multipli con tipi derivati specificati in modo dinamico

Da dove veniamo

A volte non possiamo fornire tutti i metadati richiesti necessari per il framework XmlSerializer nell'attributo. Supponiamo di avere una classe base di oggetti serializzati e alcune delle classi derivate sono sconosciute alla classe base. Non possiamo posizionare un attributo per tutte le classi che non sono conosciute al momento della progettazione del tipo di base. Potremmo avere un'altra squadra che sviluppa alcune delle classi derivate.

Cosa possiamo fare

Possiamo usare il `new XmlSerializer(type, knownTypes)` , ma sarebbe un'operazione O (N ^ 2) per i serializzatori N, almeno per scoprire tutti i tipi forniti negli argomenti:

```
// Beware of the N^2 in terms of the number of types.
var allSerializers = allTypes.Select(t => new XmlSerializer(t, allTypes));
```

```
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary (i => allTypes[i], i => allSerializers[i])
```

In questo esempio, il tipo Base non è a conoscenza dei suoi tipi derivati, che è normale in OOP.

Farlo in modo efficiente

Fortunatamente, esiste un metodo che affronta questo particolare problema, fornendo in modo efficiente tipi noti per più serializzatori:

Metodo System.Xml.Serialization.XmlSerializer.FromTypes (Tipo [])

> Il metodo FromTypes consente di creare in modo efficiente una matrice di oggetti XmlSerializer per l'elaborazione di un array di oggetti Type.

```
var allSerializers = XmlSerializer.FromTypes(allTypes);
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary(i => allTypes[i], i => allSerializers[i]);
```

Ecco un esempio di codice completo:

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

public class Program
{
    public class Container
    {
        public Base Base { get; set; }
    }

    public class Base
    {
        public int JustSomePropInBase { get; set; }
    }

    public class Derived : Base
    {
        public int JustSomePropInDerived { get; set; }
    }

    public void Main()
    {
        var sampleObject = new Container { Base = new Derived() };
        var allTypes = new[] { typeof(Container), typeof(Base), typeof(Derived) };

        Console.WriteLine("Trying to serialize without a derived class metadata:");
        SetupSerializers(allTypes.Except(new[] { typeof(Derived) }).ToArray());
        try
        {
            Serialize(sampleObject);
        }
```

```
        catch (InvalidOperationException e)
        {
            Console.WriteLine();
            Console.WriteLine("This error was anticipated,");
            Console.WriteLine("we have not supplied a derived class.");
            Console.WriteLine(e);
        }
        Console.WriteLine("Now trying to serialize with all of the type information:");
        SetupSerializers(allTypes);
        Serialize(sampleObject);
        Console.WriteLine();
        Console.WriteLine("Slides down well this time!");
    }

    static void Serialize<T>(T o)
    {
        serializerDictionary[typeof(T)].Serialize(Console.Out, o);
    }

    private static Dictionary<Type, XmlSerializer> serializerDictionary;

    static void SetupSerializers(Type[] allTypes)
    {
        var allSerializers = XmlSerializer.FromTypes(allTypes);
        serializerDictionary = Enumerable.Range(0, allTypes.Length)
            .ToDictionary(i => allTypes[i], i => allSerializers[i]);
    }
}
```

Produzione:

```
Trying to serialize without a derived class metadata:
<?xml version="1.0" encoding="utf-16"?>
<Container xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
This error was anticipated,
we have not supplied a derived class.
System.InvalidOperationException: There was an error generating the XML document. --->
System.InvalidOperationException: The type Program+Derived was not expected. Use the
XmlInclude or SoapInclude attribute to specify types that are not known statically.
   at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write2_Base(String
n, String ns, Base o, Boolean isNullable, Boolean needType)
   at
Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write3_Container(String
n, String ns, Container o, Boolean isNullable, Boolean needType)
   at
Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write4_Container(Object
o)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
   --- End of inner exception stack trace ---
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces)
   at Program.Serialize[T](T o)
   at Program.Main()
Now trying to serialize with all of the type information:
```

```
<?xml version="1.0" encoding="utf-16"?>
<Container xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Base xsi:type="Derived">
    <JustSomePropInBase>0</JustSomePropInBase>
    <JustSomePropInDerived>0</JustSomePropInDerived>
  </Base>
</Container>
Slides down well this time!
```

Cosa c'è nell'output

Questo messaggio di errore consiglia ciò che abbiamo cercato di evitare (o cosa non possiamo fare in alcuni scenari) - fare riferimento ai tipi derivati dalla classe base:

```
Use the XmlInclude or SoapInclude attribute to specify types that are not known statically.
```

Ecco come otteniamo la nostra classe derivata nell'XML:

```
<Base xsi:type="Derived">
```

`Base` corrisponde al tipo di proprietà dichiarato nel tipo `Container` e `Derived` è il tipo dell'istanza effettivamente fornita.

Ecco un esempio pratico di violino

Leggi XmlSerializer online: https://riptutorial.com/it/dot-net/topic/31/xmlserializer

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con .NET Framework	Adriano Repetti, Alan McBee, ale10ander, Andrew Jens, Andrew Morton, Andrey Shchekin, Community, Daniel A. White, Ehsan Sajjad, harriyott, hillary.fraley, Ian, James Thorpe, Jamie Rees, Joel Martinez, Kevin Montrose, Lirrik, MarcinJuraszek, matteeyah, naveen, Nicholas Sizer, Paweł Izdebski, Peter, Peter Gordon, Peter Hommel, PSN, Richard Lander, Rion Williams, Robert Columbia, RubberDuck, SeeuD1, Serg Rogovtsev, Squidward, Stephen Leppik, Steven Doggart, svick, ɬolɐəz əuɬ qoq
2	.NET Core	Mihail Stancescu
3	ADO.NET	Akshay Anand, Andrew Morton, Daniel A. White, DavidG, Drew, elmer007, Hamid, Harjot, Heinzi, Igor, user2321864
4	Alberi di espressione	Akshay Anand, George Polevoy, Jim, n.podbielski, Pavel Mayorov, RamenChef, Stephen Leppik, Stilgar, wangengzheng
5	Carica file e dati POST sul server web	Aleks Andreev
6	Client HTTP	CodeCaster, Konamiman, MuiBienCarlota
7	CLR	Gajendra, starbeamrainbowlabs, Theodoros Chatzigiannakis
8	collezioni	Alan McBee, Aman Sharma, Anik Saha, Daniel A. White, demonplus, Felipe Oriani, harriyott, Ian, Mark C., Ravi A., Virtlink
9	Compilatore JIT	Krikor Ailanjian
10	Contesti di sincronizzazione	DLeh, Gusdor
11	Contratti di codice	JJS, Matthew Whited, RamenChef
12	Crittografia / Crittografia	Alexander Mandt, Daniel A. White,

		demonplus, Jagadisha B S, lokusking, Matt
13	Data e ora di analisi	GalacticCowboy, John
14	dizionari	Adriano Repetti, Bjørn-Roger Kringsjå, Daniel Plaisted, Darrel Lee, Felipe Oriani, George Duckett, George Polevoy, hatchet, Hogan, Ian, LegionMammal978, Luke Bearl, Olivier Jacot-Descombes, RamenChef, Ringil, Robert Columbia, Stephen Byrne, the berserker, Tomáš Hübelbauer
15	eccezioni	Adi Lester, Akshay Anand, Alan McBee, Alfred Myers, Arvin Baccay, BananaSft, CodeCaster, Dave R., Kritner, Mafii, Matt, Rob, Sean, starbeamrainbowlabs, STW, Yousef Al-Mulla
16	Elaborazione parallela mediante framework .Net	Yahfoufi
17	Espressioni regolari (System.Text.RegularExpressions)	BrunoLM, Denuath, Matt dc, tehDorf
18	File Input / Output	ale10ander, Alexander Mandt, Ingenioushax, Nitram
19	Gestione della memoria	Big Fan, binki, DrewJordan
20	Globalizzazione in ASP.NET MVC tramite l'internazionalizzazione intelligente per ASP.NET	Scott Hannen
21	Glossario degli acronimi	Tanveer Badar
22	Impostazione affinità processo e discussione	MSE, RamenChef
23	impostazioni	Alan McBee
24	Iniezione di dipendenza	Phil Thomas, Scott Hannen
25	JSON in .NET con Newtonsoft.Json	DLeh
26	Lavora con SHA1 in C #	mahdi abasi
27	Lettura e scrittura di file zip	Arxae
28	LINQ	A. Raza, Adil Mammadov, Akshay Anand, Alexander V., Benjamin Hodgson,

		Blachshma, Bradley Grainger, Bruno Garcia, Carlos Muñoz, CodeCaster, dbasnett, DoNot , dotctor, Eduardo Molteni, Ehsan Sajjad, GalacticCowboy, H. Pauwelyn, Haney, J3soon, jbtule, jnovo, Joe Amenta, Kilazur, Konamiman, MarcinJuraszek, Mark Hurd, McKay, Mellow, Mert Gülsoy, Mike Stortz, Mr.Mindor, Nate Barbettini, Pavel Voronin, Ruben Steins, Salvador Rubio Martinez, Sammi, Sergio Domínguez, Sidewinder94
29	Managed Extensibility Framework	Joe Amenta, Kirk Broadhurst, RamenChef
30	Moduli VB	ale10ander, dbasnett
31	Networking	Konamiman
32	Panoramiche API Task Parallel Library (TPL)	Gusdor, Jacobr365
33	Per ciascuno	Dr Rob Lang, just.ru, Lucas Trzesniewski
34	Platform Invoke	Dmitry Egorov, Imran Ali Khan
35	Porte seriali	Dmitry Egorov
36	Raccolta dei rifiuti	avat
37	ReadOnlyCollections	tehDorf
38	Riflessione	Aleks Andreev, Bjørn-Roger Kringsjå, demonplus, Jean-Baptiste Noblot, Jigar, JJP , Kirk Broadhurst, Lorenzo Dematté, Matas Vaitkevicius, NetSquirrel, Pavel Mayorov, Peter, smdrager, Terry, user1304444, void
39	Scrivi e leggi dallo stream di StdErr	Aleks Andreev
40	Serializzazione JSON	Akshay Anand, Andrius, Eric, hasan, M22an, PedroSouki, Thriggle, Tolga Evcimen
41	Server HTTP	Devon Burriss, Konamiman
42	Sistema di imballaggio NuGet	Andrey Shchekin, Anik Saha, Ashtonian, CodeCaster, Daniel A. White, Matas Vaitkevicius, Ozair Kafray
43	SpeechRecognitionEngine classe per riconoscere il parlato	ProgramFOX, RamenChef

44	Stack e Heap	Hywel Rees
45	stringhe	Adriano Repetti, Alexander Mandt, Matt, Pavel Voronin, RamenChef
46	System.Diagnostics	Adi Lester, Bassie, Fredou, Ogglas, Ondřej Štorc, RamenChef
47	System.IO	CodeCaster, Daniel A. White, demonplus, Filip Frącz, RoyalPotato
48	System.IO.File class	Adriano Repetti, delete me
49	System.Net.Mail	demonplus, Steve, vicky
50	System.Reflection.Emit spazio dei nomi	Luaan, NikolayKondratyev, RamenChef, toddmo
51	System.Runtime.Caching.MemoryCache (ObjectCache)	Guanxi, RamenChef
52	Task Parallel Library (TPL)	Adi Lester, Aman Sharma, Andrew, i3arnon, Jacobr365, JamyRyals, Konamiman, Mathias Müller, Mert Gülsoy, Mikhail Filimonov, Pavel Mayorov, Pavel Voronin, RamenChef, Thomas Bledsoe, TorbenJ
53	Test unitario	Axarydax
54	threading	Behzad, Martijn Pieters, Mellow
55	Tipi personalizzati	Alan McBee, DrewJordan, matteeyah
56	TPL Dataflow	i3arnon, Jacobr365, Nikola.Lukovic, RamenChef
57	Usando il progresso e IProgress	DLeh
58	XmlSerializer	Aphelion, George Polevoy, RamenChef, Rowland Shaw, Thomas Levesque, void, Yogi