using System;

using System.Linq;

class Program

{

static void Main()

{

string[] words = {"hello", "wonderful", "LINQ", "beautiful", "world"};

//Get only short words

var shortWords = **from word in words where word.Length <= 5 select word**;

//Print each word out

**foreach (var word in shortWords)**

{

Console.WriteLine(word);

}

Console.ReadLine();

}

}

**Lamda Syntax**

var longWords = words.Where( w => w.length > 10);

**Query Syntax**

var longwords = from w in words where w.length > 10;

LINQ tiene una arquitectura de tres capas en la que la capa superior consta de las extensiones de lenguaje y la capa inferior consta de fuentes de datos que son típicamente objetos que implementan interfaces genéricas IEnumerable <T> o IQueryable <T>. La arquitectura se muestra a continuación.



**Query Expressions**

La expresión de consulta no es otra cosa que una consulta LINQ, expresada en una forma similar a la de SQL con operadores de consulta como Select, Where y OrderBy. Las expresiones de consulta usualmente comienzan con la palabra clave "from".

Para acceder a operadores de consulta LINQ estándar, el espacio de nombres System.Query se debe importar de forma predeterminada.

A continuación se muestra un ejemplo para mostrar una operación de consulta completa que consiste en creación de origen de datos, definición de expresión de consulta y ejecución de consulta.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Operators

{

class LINQQueryExpressions

{

static void Main()

{

// Specify the data source.

int[] scores = new int[] { 97, 92, 81, 60 };

// Define the query expression.

IEnumerable<int> scoreQuery = **from score in scores where score > 80 select score;**

// Execute the query.

**foreach (int i in scoreQuery)**

**{**

Console.Write(i + " ");

}

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result:

97 92 81

**Diferencia entre LINQ y procedimiento almacenado**

Hay una matriz de diferencias existentes entre LINQ y procedimientos almacenados. Estas diferencias se mencionan a continuación.

 Los procedimientos almacenados son mucho más rápidos que una consulta LINQ ya que siguen un plan de ejecución esperado.

 Es fácil evitar errores en tiempo de ejecución al ejecutar una consulta LINQ en comparación con un procedimiento almacenado ya que el primero tiene soporte de Intellisense de Visual Studio, así como la comprobación de tipo completo durante la compilación.

 LINQ permite la depuración mediante el uso de depurador .NET que no es en el caso de procedimientos almacenados.

 LINQ ofrece soporte para múltiples bases de datos en contraste con los procedimientos almacenados, donde es esencial volver a escribir el código para diversos tipos de bases de datos.

 La implementación de una solución basada en LINQ es fácil y sencilla en comparación con la implementación de un conjunto de procedimientos almacenados

**Necesidad de LINQ**

Antes de LINQ, era esencial para aprender C #, SQL, y varias API que unen los dos lenguajes para formar una aplicación completa.

A continuación se muestra un ejemplo de cuántas diversas técnicas fueron utilizadas por los desarrolladores al consultar un dato antes de la llegada de LINQ.

SqlConnection sqlConnection = new SqlConnection(connectString);

SqlConnection.Open();

System.Data.SqlClient.SqlCommand sqlCommand = new SqlCommand();

sqlCommand.Connection = sqlConnection;

sqlCommand.CommandText = "Select \* from Customer";

return sqlCommand.ExecuteReader (CommandBehavior.CloseConnection)

Curiosamente, fuera de las líneas de código ofrecidas, la consulta se define sólo por los dos últimos. Usando LINQ, la misma consulta de datos se puede escribir en un formato legible codificado por colores como el siguiente mencionado más adelante que también en un tiempo muy menor.

Northwind db = new Northwind(@"C:\Data\Northwnd.mdf");

var query = from c in db.Customers select c;

**Ventajas de LINQ**

LINQ ofrece una gran cantidad de ventajas y entre ellas la principal es su poderosa expresividad que permite a los desarrolladores expresar declarativamente. Algunas de las otras ventajas de LINQ se dan a continuación.

 LINQ ofrece resaltado de sintaxis que resulta útil para detectar errores durante el tiempo de diseño.

 LINQ ofrece IntelliSense, lo que significa escribir consultas más precisas con facilidad.

 Escribir códigos es bastante más rápido en LINQ y así el tiempo de desarrollo también se reduce significativamente.

 LINQ facilita la depuración debido a su integración en el lenguaje C #.

 Ver la relación entre dos tablas es fácil con LINQ debido a su característica jerárquica y esto permite componer consultas que unen múltiples tablas en menos tiempo.

 LINQ permite el uso de una sintaxis LINQ única mientras se consulta muchas fuentes de datos diversas.

 LINQ es extensible, lo que significa que es posible utilizar el conocimiento de LINQ para consultar nuevos tipos de fuentes de datos.

 LINQ ofrece la posibilidad de unir varias fuentes de datos en una única consulta, además de romper problemas complejos en un conjunto de consultas cortas fáciles de depurar.

 LINQ ofrece una fácil transformación para la conversión de un tipo de datos a otro, como la transformación de datos SQL en datos XML.

**Standard Query Operators**

Un conjunto de métodos de extensión que forman un patrón de consulta se conoce como LINQ Standard Query Operators. Como bloques de construcción de expresiones de consulta de LINQ, estos operadores ofrecen una gama de capacidades de consulta como filtrado, clasificación, agregación, etc.

Los operadores de consulta estándar de LINQ se pueden clasificar en los siguientes en función de su funcionalidad.

Filtering Operators

 Join Operators

 Projection Operations

 Sorting Operators

 Grouping Operators

 Conversions

 Concatenation

 Aggregation

 Quantifier Operations

 Partition Operations

 Generation Operations

 Set Operations

 Equality

 Element Operators

**Operadores de Filtrado**

El filtrado es una operación para restringir el conjunto de resultados de manera que sólo haya seleccionado elementos que satisfagan una condición particular.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **C# Query Expression Syntax** | **VB Query Expression Syntax** |
| Where | Filter values based on a predicate function | Where | Where |

|  |  |  |  |
| --- | --- | --- | --- |
| OfType | Filter values based on their ability to be as a specified type | Not Applicable | Not Applicable |

**Example of Where − Query Expression**

**C#**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Operators

{

class Program

{

static void Main(string[] args)

{

string[] words = { "humpty", "dumpty", "set", "on", "a", "wall" };

IEnumerable<string> query = **from word in words where word.Length == 3 select word;**

foreach (string str in query)

Console.WriteLine(str);

Console.ReadLine();

}

}

}

**Operadores Join**

Unirse se refiere a una operación en la que se dirigen las fuentes de datos con relaciones difíciles de seguir entre sí de una manera directa.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **C# Query Expression Syntax** | **VB Query Expression Syntax** |
| Join | The operator join two sequences on basis of matching keys | join … in … on … equals … | From x In …, y In … Where x.a = y.a |

|  |  |  |  |
| --- | --- | --- | --- |
| GroupJoin | Join two sequences and group the matching elements | join … in … on … equals … into … | Group Join … In … On … |

**Example of Join − Query Expression**

**C#**

using System;

using System.Collections.Generic;

using System.Linq;

namespace Operators

{

class JoinTables

{

**class DepartmentClass**

{

public int DepartmentId { get; set; }

public string Name { get; set; }

}

**class EmployeeClass**

{

public int EmployeeId { get; set; }

public string EmployeeName { get; set; }

public int DepartmentId { get; set; }

}

static void Main(string[] args)

{

List <DepartmentClass> departments = new List <DepartmentClass>();

departments.Add(new DepartmentClass { DepartmentId = 1, Name = "Account" });

departments.Add(new DepartmentClass { DepartmentId = 2, Name = "Sales" });

departments.Add(new DepartmentClass { DepartmentId = 3, Name = "Marketing" });

List <EmployeeClass> employees = new List <EmployeeClass>();

employees.Add(new EmployeeClass { DepartmentId = 1, EmployeeId = 1, EmployeeName = "William" });

employees.Add(new EmployeeClass { DepartmentId = 2, EmployeeId = 2, EmployeeName = "Miley" });

employees.Add(new EmployeeClass { DepartmentId = 1, EmployeeId = 3, EmployeeName = "Benjamin" });

var list = (from e **in** **employees** **join** **d** **in departments** **on** **e.DepartmentId** **equals** **d.DepartmentId**

select **new**

**{**

**EmployeeName = e.EmployeeName,**

**DepartmentName = d.Name**

**});**

foreach (var e in list)

{

Console.WriteLine("Employee Name = {0} , Department Name = {1}", e.EmployeeName, e.DepartmentName);

}

Console.WriteLine("\nPress any key to continue.");

Console.ReadKey();

}

}

}

**Projection Operations**

|  |  |  |  |
| --- | --- | --- | --- |
| Projection is an operation in which an object is transformed into an altogether new form with only specific properties. **Operator** | **Description** | **C# Query Expression Syntax** | **VB Query Expression Syntax** |
| Select | The operator projects values on basis of a transform function | select | Select |
| SelectMany | The operator project the sequences of values which are based on a transform function as well as flattens them into a single sequence | Use multiple from clauses | Use multiple From clauses |

**Example of Select − Query Expression**

**C#**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Operators

{

class Program

{

static void Main(string[] args)

{

List<string> words = new List<string>() { "an", "apple", "a", "day" };

var query = **from word in words select word.Substring(0, 1);**

foreach (string s in query)

Console.WriteLine(s);

Console.ReadLine();

}

}

}

**Example of SelectMany − Query Expression**

**C#**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Operators

{

class Program

{

static void Main(string[] args)

{

List<string> phrases = new List<string>() { "an apple a day", "the quick brown fox" };

**var query = from phrase in phrases**

**from word in phrase.Split(' ')**

**select word;**

foreach (string s in query)

Console.WriteLine(s);

Console.ReadLine();

}

}

}

**Sorting Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| A sorting operation allows ordering the elements of a sequence on basis of a single or more attributes. **Operator** | **Description** | **C# Query Expression Syntax** | **VB Query Expression Syntax** |
| OrderBy | The operator sort values in an ascending order | orderby | Order By |
| OrderByDescending | The operator sort values in a descending order | orderby ... descending | Order By ... Descending |
| ThenBy | Executes a secondary sorting in an ascending order | orderby …, … | Order By …, … |
| ThenByDescending | Executes a secondary sorting in a descending order | orderby …, … descending | Order By …, … Descending |
| Reverse | Performs a reversal of the order of the elements in a collection | Not Applicable | Not Applicable |

**Example of OrderBy, OrderByDescending − Query Expression**

**C#**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Operators

{

class Program

{

static void Main(string[] args)

{

int[] num = { -20, 12, 6, 10, 0, -3, 1 };

//create a query that obtain the values in sorted order

var posNums = **from n in num**

**orderby n**

**select n;**

Console.Write("Values in ascending order: ");

// Execute the query and display the results.

foreach (int i in posNums)

Console.Write(i + " \n");

var posNumsDesc = from n in num

orderby n descending

select n;

Console.Write("\nValues in descending order: ");

// Execute the query and display the results.

foreach (int i in posNumsDesc)

Console.Write(i + " \n");

Console.ReadLine();

}

}

}

**Grouping Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| The operators put data into some groups based on a common shared attribute. **Operator** | **Description** | **C# Query Expression Syntax** | **VB Query Expression Syntax** |
| GroupBy | Organize a sequence of items in groups and return them as an IEnumerable collection of type IGrouping<key, element> | group … by -or- group … by … into … | Group … By … Into … |
| ToLookup | Execute a grouping operation in which a sequence of key pairs are returned | Not Applicable | Not Applicable |

**Example of GroupBy − Query Expression**

**C#**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Operators

{

class Program

{

static void Main(string[] args)

{

List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199, 329, 446, 208 };

**IEnumerable<IGrouping<int, int>> query = from number in numbers**

**group number by number % 2;**

foreach (var group in query)

{

Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");

foreach (int i in group)

Console.WriteLine(i);

}

Console.ReadLine();

}

}

}

**Conversions**

|  |  |  |  |
| --- | --- | --- | --- |
| The operators change the type of input objects and are used in a diverse range of applications. **Operator** | **Description** | **C# Query Expression Syntax** | **VB Query Expression Syntax** |
| AsEnumerable | Returns the input typed as IEnumerable<T> | Not Applicable | Not Applicable |
| AsQueryable | A (generic) IEnumerable is converted to a (generic) IQueryable | Not Applicable | Not Applicable |
| Cast | Performs casting of elements of a collection to a specified type | Use an explicitly typed range variable. Eg: from string str in words | From … As … |

|  |  |  |  |
| --- | --- | --- | --- |
| OfType | Filters values on basis of their , depending on their capability to be cast to a particular type | Not Applicable | Not Applicable |
| ToArray | Forces query execution and does conversion of a collection to an array | Not Applicable | Not Applicable |
| ToDictionary | On basis of a key selector function set elements into a Dictionary<TKey, TValue> and forces execution of a LINQ query | Not Applicable | Not Applicable |
| ToList | Forces execution of a query by converting a collection to a List<T> | Not Applicable | Not Applicable |
| ToLookup | Forces execution of a query and put elements into a Lookup<TKey, TElement> on basis of a key selector function | Not Applicable | Not Applicable |

**Example of Cast − Query Expression**

**C#**

using System;

using System.Linq;

namespace Operators

{

class Cast

{

static void Main(string[] args)

{

Plant[] plants = new Plant[] {new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },

new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },

new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },

new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }};

**var query = from CarnivorousPlant cPlant in plants**

**where cPlant.TrapType == "Snap Trap"**

**select cPlant;**

foreach (var e in query)

{

Console.WriteLine("Name = {0} , Trap Type = {1}", e.Name, e.TrapType);

}

Console.WriteLine("\nPress any key to continue.");

Console.ReadKey();

}

}

class Plant

{

public string Name { get; set; }

}

class CarnivorousPlant : Plant

{

public string TrapType { get; set; }

}

}

ENTITIES



**Example of ADD, UPDATE, and DELETE using LINQ with Entity Model**

using System;

using System.Linq;

namespace LINQTOSQLConsoleApp

{

public class LinqToEntityModel

{

static void Main(string[] args)

{

**using (LinqToSQLDBEntities context = new LinqToSQLDBEntities())**

**{**

**//Get the List of Departments from Database**

**var departmentList = from d in context.Departments**

**select d;**

**foreach (var dept in departmentList)**

**{**

Console.WriteLine("Department Id = {0} , Department Name = {1}", dept.DepartmentId, dept.Name);

}

**//Add new Department**

DataAccess.Department department = new DataAccess.Department();

department.Name = "Support";

**context.Departments.Add(department);**

**context.SaveChanges();**

Console.WriteLine("Department Name = Support is inserted in Database");

**//Update existing Department**

**DataAccess.Department updateDepartment = context.Departments.FirstOrDefault(d ⇒ d.DepartmentId == 1);**

**updateDepartment.Name = "Account updated";**

**context.SaveChanges();**

Console.WriteLine("Department Name = Account is updated in Database");

**//Delete existing Department**

**DataAccess.Department deleteDepartment = context.Departments.FirstOrDefault(d ⇒ d.DepartmentId == 3);**

**context.Departments.Remove(deleteDepartment);**

**context.SaveChanges();**

Console.WriteLine("Department Name = Pre-Sales is deleted in Database");

//Get the Updated List of Departments from Database

departmentList = from d in context.Departments

select d;

foreach (var dept in departmentList)

{

Console.WriteLine("Department Id = {0} , Department Name = {1}", dept.DepartmentId, dept.Name);

}

}

Console.WriteLine("\nPress any key to continue.");

Console.ReadKey();

}

}

}

LAMBDA EXPRESSIONS

El término «expresión Lambda» ha derivado su nombre del cálculo «lambda», que a su vez es una notación matemática aplicada para definir las funciones. Las expresiones Lambda como parte ejecutable de una ecuación LINQ traducen la lógica de una manera en tiempo de ejecución para que pueda pasar a la fuente de datos convenientemente. Sin embargo, las expresiones lambda no sólo se limitan a buscar la aplicación en LINQ solamente.

Estas expresiones se expresan mediante la siguiente sintaxis:

**(Input parameters) ⇒ Expression or statement block**

Aquí hay un ejemplo de una expresión lambda –

**y ⇒ y \* y**

A continuación se muestra un ejemplo de una expresión lambda en C #.

System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace lambdaexample

{

class Program

{

**delegate int del(int i);**

static void Main(string[] args)

{

**del myDelegate = y ⇒ y \* y;**

**int j = myDelegate(5);**

Console.WriteLine(j);

Console.ReadLine();

}

}

}

**Expresión Lambda**

Como la expresión en la sintaxis de la expresión lambda mostrada arriba está en el lado derecho, éstos también se conocen como expresión lambda.

**Async Lambdas**

La expresión lambda creada mediante la incorporación de procesamiento asíncrono mediante el uso de palabras clave asíncronas se conoce como lambdas asíncrona. A continuación se muestra un ejemplo de lambda asíncrona.

Func<Task<string>> getWordAsync = async() ⇒ “hello”;

**Lambda en Operadores de Consulta Estándar**

Una expresión lambda dentro de un operador de consulta es evaluada por el mismo a petición y trabaja continuamente en cada uno de los elementos en la secuencia de entrada y no en toda la secuencia. Los desarrolladores están autorizados por la expresión de Lambda para alimentar su propia lógica en los operadores de consulta estándar. En el ejemplo siguiente, el desarrollador ha utilizado el operador 'Where' para recuperar los valores impares de la lista dada mediante el uso de una expresión lambda.

/Get the average of the odd Fibonacci numbers in the series...

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace lambdaexample

{

class Program

{

static void Main(string[] args)

{

int[] fibNum = { 1, 1, 2, 3, 5, 8, 13, 21, 34 };

double averageValue = **fibNum.Where(num => num % 2 == 1).Average();**

Console.WriteLine(averageValue);

Console.ReadLine();

}

}

}

**Inferencia de Tipo en Lambda**

En C #, la inferencia de tipo se utiliza convenientemente en una variedad de situaciones y que también sin especificar los tipos explícitamente. Sin embargo, en el caso de una expresión lambda, la inferencia de tipo funcionará sólo cuando cada tipo ha sido especificado como el compilador debe ser satisfecho. Consideremos el siguiente ejemplo.

Delegate int Transformer (int i);

Aquí el compilador utiliza la inferencia de tipo para basarse en el hecho de que x es un entero y esto se hace examinando el tipo de parámetro del transformador.

**Ámbito Variable en la Expresión Lambda**

Hay algunas reglas mientras se utiliza el ámbito de variables en una expresión lambda como las variables que se inician dentro de una expresión lambda no están destinadas a ser visibles en un método externo. También existe una regla de que una variable capturada no debe ser recogida de basura a menos que el delegado que haga referencia a la misma sea elegible para la recolección de basura. Además, existe una regla que prohíbe una declaración de retorno dentro de una expresión lambda para provocar la devolución de un método .

Este es un ejemplo para demostrar el alcance variable en la expresión lambda.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace lambdaexample

{

class Program

{

delegate bool D();

delegate bool D2(int i);

class Test

{

D del;

D2 del2;

public void TestMethod(int input)

{

int j = 0;

// Initialize the delegates with lambda expressions.

// Note access to 2 outer variables.

// del will be invoked within this method.

del = () ⇒ { j = 10; return j > input; };

// del2 will be invoked after TestMethod goes out of scope.

del2 = (x) ⇒ { return x == j; };

// Demonstrate value of j:

// The delegate has not been invoked yet.

Console.WriteLine("j = {0}", j); // Invoke the delegate.

bool boolResult = del();

Console.WriteLine("j = {0}. b = {1}", j, boolResult);

}

static void Main()

{

Test test = new Test();

test.TestMethod(5);

// Prove that del2 still has a copy of

// local variable j from TestMethod.

bool result = test.del2(10);

Console.WriteLine(result);

Console.ReadKey();

}

}

}

}

When the above code is compiled and executed, it produces the following result:

j = 0

j = 10. b = True

True

**Expression Tree**

Lambda expressions are used in **Expression Tree** construction extensively. An expression tree give away code in a data structure resembling a tree in which every node is itself an expression like a method call or can be a binary operation like x<y. Below is an example of usage of lambda expression for constructing an expression tree.

**Statement Lambda**

There is also **statement lambdas** consisting of two or three statements, but are not used in construction of expression trees. A return statement must be written in a statement lambda.

Syntax of statement lambda

(params) ⇒ {statements}

**Example of a statement lambda**

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Linq.Expressions;

namespace lambdaexample

{

class Program

{

static void Main(string[] args)

{

int[] source = new[] { 3, 8, 4, 6, 1, 7, 9, 2, 4, 8 };

foreach (int i in source.Where(x ⇒

{

if (x <= 3)

return true;

else if (x >= 7)

return true;

return false;

}

))

Console.WriteLine(i);

Console.ReadLine();

}

}

}

Lambdas are employed as arguments in LINQ queries based on methods. They are never allowed to have a place on the left side of operators like **is** or **as,** just like anonymous methods. Although, Lambda expressions are much alike anonymous methods, these are not at all restricted to be used as delegates only.

**Points to remember while using lambda expressions**

 A lambda expression can return a value and may have parameters.

 Parameters can be defined in a myriad of ways with a lambda expression.

 If there is single statement in a lambda expression, there is no need of curly brackets whereas if there are multiple statements, curly brackets as well as return value are essential to write.

 With lambda expressions, it is possible to access variables present outside of the lambda expression block by a feature known as closure. Use of closure should be done cautiously to avoid any problem.

 It is impossible to execute any unsafe code inside any lambda expression.

 Lambda expressions are not meant to be used on the operator’s left side.