# Tutorial LINQ

Language-Integrated Query (LINQ) es un potente lenguaje de consulta introducido con .Net 3.5 y Visual Studio 2008. LINQ se puede usar con C # o Visual Basic para consultar diferentes fuentes de datos.

Los tutoriales de LINQ le ayudarán a aprender el lenguaje LINQ utilizando temas que van desde lo básico a lo avanzado. Estos tutoriales se desglosan en series de temas relacionados, de modo que se parte de un tema que debe ser entendido primero, y luego se aprenden gradualmente otras características de LINQ secuencialmente. Los tutoriales de LINQ están repletos de explicaciones fáciles de entender, ejemplos del mundo real, consejos útiles, notas informativas y puntos para recordar.

## ¿Para quien?

Estos tutoriales están diseñados para principiantes y profesionales que quieran aprender LINQ paso a paso.

## Prerrequisitos:

Se requiere conocimiento básico de .Net Framework 3.5 / 4.5, C #, Visual Studio.

## Examen

Pon a prueba tu conocimiento de LINQ - [Inicia prueba LINQ](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/online-test/linq-test&usg=ALkJrhjQGKaOmQCHxhfLY_4ywdegsnZVWA)

Comencemos haciendo clic en Siguiente.

Que es LINQ?

LINQ (Language Integrated Query) es una sintaxis de consulta uniforme en C # y VB.NET utilizada para guardar y recuperar datos de diferentes fuentes. Está integrado en C # o VB, lo que elimina la falta de correspondencia entre los lenguajes de programación y las bases de datos, y proporciona una interfaz de consulta única para diferentes tipos de fuentes de datos.

Por ejemplo, SQL es un lenguaje de consulta estructurado que se usa para guardar y recuperar datos de una base de datos. De la misma manera, LINQ es una sintaxis de consulta estructurada construida en C # y VB.NET utilizada para guardar y recuperar datos de diferentes tipos de fuentes de datos, como una colección de objetos, una base de datos SQL server, XML, un servicio web, etc.

LINQ siempre trabaja con objetos para que pueda usar los mismos patrones básicos de codificación para consultar y transformar datos en documentos XML, bases de datos SQL, conjuntos de datos ADO.NET, colecciones .NET y cualquier otro formato para el que esté disponible un proveedor LINQ.

[](http://www.tutorialsteacher.com/Content/images/linq/linq-usage.PNG)Uso de LINQ

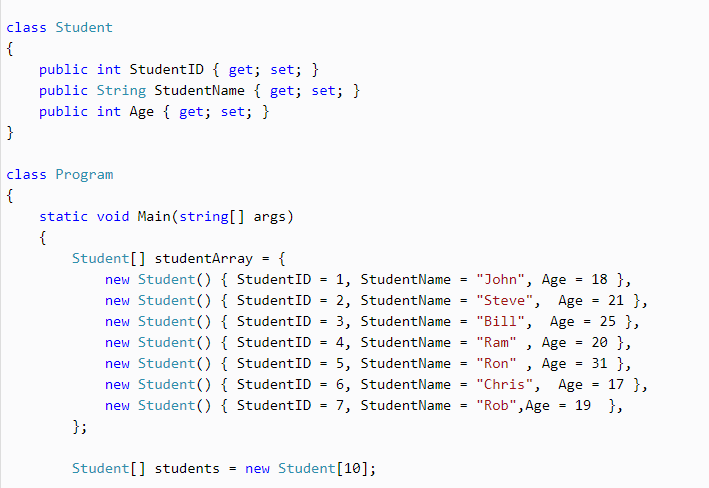
Aprenda por qué deberíamos usar LINQ en la siguiente sección.

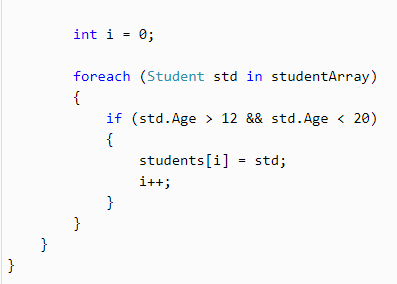
# Por qué LINQ?

Para entender por qué deberíamos usar LINQ, veamos algunos ejemplos. Supongamos que quiere encontrar una lista de estudiantes adolescentes de una variedad de objetos de Estudiante.

Antes de C # 2.0, teníamos que usar un bucle 'foreach' o 'for' para recorrer la colección y encontrar un objeto en particular. Por ejemplo, tuvimos que escribir el siguiente código para encontrar todos los objetos de estudiantes de una matriz de estudiantes, donde la edad es entre 12 y 20 (para adolescentes de 13 a 19):

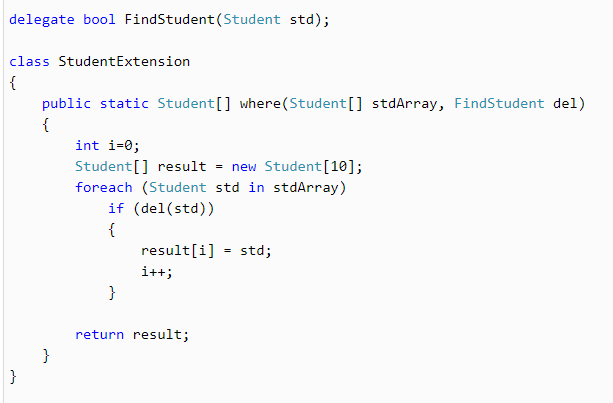
Ejemplo: Use for loop para encontrar elementos de la colección en C # 1.0

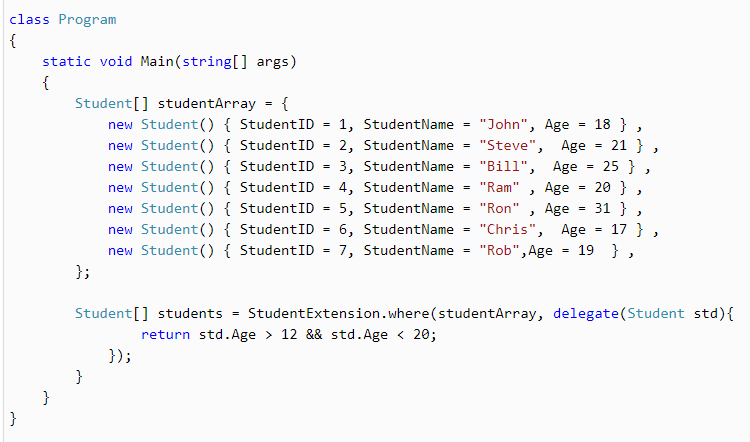




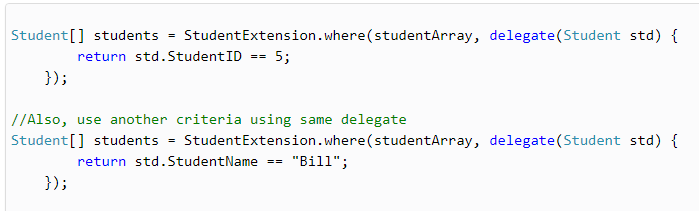
El uso del bucle for es engorroso, no se puede mantener y leer. C # 2.0 introdujo el **delegado** , que se puede usar para manejar este tipo de escenario, como se muestra a continuación.

Ejemplo: utilice delegados para buscar elementos de la colección en C # 2.0





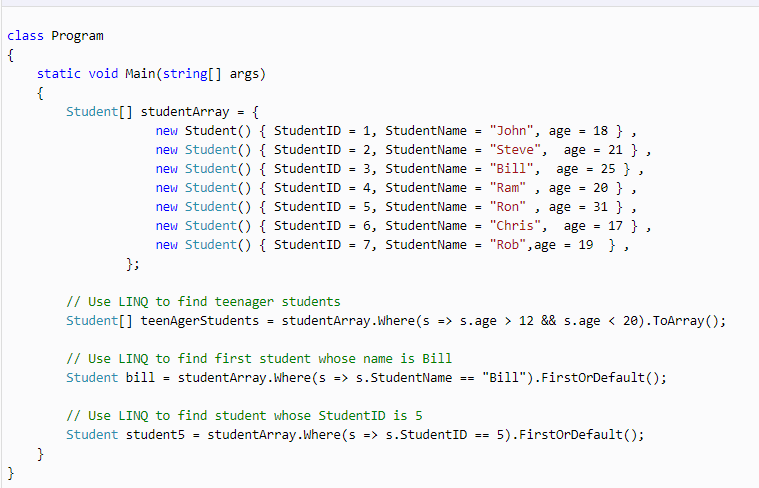
Entonces, con C # 2.0, tiene la ventaja de **delegar** en encontrar estudiantes con cualquier criterio. No tiene que usar un ciclo for para encontrar estudiantes que usan diferentes criterios. Por ejemplo, puede usar la misma función de delegado para encontrar un estudiante cuyo ID de alumno sea 5 o cuyo nombre sea Bill, como se muestra a continuación:



El equipo de C # sintió que aún necesitaban hacer el código aún más compacto y legible. Entonces introdujeron el método de extensión, la expresión lambda, el árbol de expresiones, el tipo anónimo y la expresión de consulta en [C # 3.0](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/csharp/csharp-version-history&usg=ALkJrhg86YTOdvdkPPQ_A0DwN40Yl-W2eg) . Puede usar estas características de C # 3.0, que son bloques de construcción de LINQ para consultar los diferentes tipos de colección y obtener los elementos resultantes en una sola declaración.

El siguiente ejemplo muestra cómo puede usar la consulta LINQ con la expresión lambda para encontrar un alumno (s) en particular de la colección de estudiantes.

C # 3.0 en adelante:



Como puede ver en el ejemplo anterior, especificamos diferentes criterios utilizando el operador LINQ y la expresión lambda en una sola instrucción. Por lo tanto, LINQ hace que el código sea más compacto y legible, y también se puede usar para consultar diferentes fuentes de datos. Por ejemplo, si tiene una tabla de estudiantes en una base de datos en lugar de una matriz de objetos estudiantiles como se indicó anteriormente, puede seguir utilizando la misma consulta para buscar estudiantes que utilicen [Entity Framework](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.entityframeworktutorial.net/&usg=ALkJrhjtu4KynxzAXC__kBOGjSI_t3Fj5A) .

## Ventajas de LINQ:

* **Lenguaje familiar: los** desarrolladores no tienen que aprender un nuevo lenguaje de consulta para cada tipo de fuente de datos o formato de datos.
* **Menos codificación:** reduce la cantidad de código que se debe escribir en comparación con un enfoque más tradicional.
* **Código** legible **:** LINQ hace que el código sea más legible para que otros desarrolladores puedan entenderlo y mantenerlo fácilmente.
* **Forma estandarizada de consultar múltiples fuentes de datos:** la misma sintaxis LINQ se puede utilizar para consultar múltiples fuentes de datos.
* **Compilar la seguridad del tiempo de las consultas:** proporciona verificación de tipos de objetos en tiempo de compilación.
* **Soporte IntelliSense:** LINQ proporciona IntelliSense para colecciones genéricas.
* **Dar forma a los datos:** puede recuperar datos en diferentes formas.

# LINQ API

LINQ no es más que la colección de métodos de extensión para las clases que implementan la interfaz IEnumerable y IQueryable. [*El*](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://msdn.microsoft.com/en-us/library/system.linq(v%3Dvs.110).aspx&usg=ALkJrhjq8hqE95FesASQhQ3BvXp_Ye3hSw) espacio de nombres [*System.Linq*](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://msdn.microsoft.com/en-us/library/system.linq(v%3Dvs.110).aspx&usg=ALkJrhjq8hqE95FesASQhQ3BvXp_Ye3hSw) incluye las clases y las interfaces necesarias para LINQ. [Enumerable](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://msdn.microsoft.com/en-us/library/system.linq.enumerable(v%3Dvs.110).aspx)&usg=ALkJrhg0tGcRpDx5jN8vkVh774JMs7eUlA) y [Queryable](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://msdn.microsoft.com/en-us/library/system.linq.queryable(v%3Dvs.110).aspx&usg=ALkJrhi9Gk7_7URG9IsWhK5sUTkcEWNdRg) son dos clases estáticas principales de LINQ API que contienen métodos de extensión.

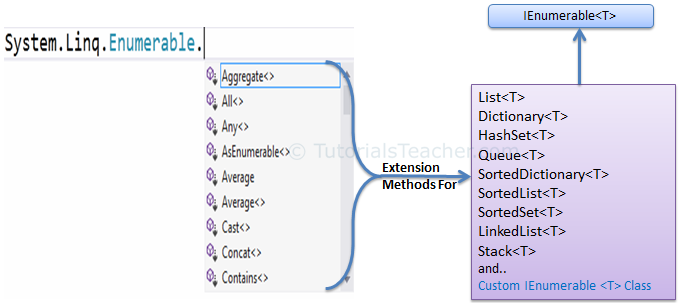
http://www.tutorialsteacher.com/Content/images/tips.png

*El* espacio de nombres *System.Linq* se incluye por defecto cuando agrega una nueva clase en Visual Studio, de modo que puede usar LINQ de forma predeterminada.

## Enumerable:

**La** clase **enumerable** incluye métodos de extensión para las clases que implementan la [interfaz IEnumerable <T>](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://msdn.microsoft.com/en-us/library/9eekhta0(v%3Dvs.110).aspx&usg=ALkJrhiG2Cj_CrRUKNmYv453Dabh0dtJFA) , esto incluye todos los tipos de colección en [System.Collections.Generic](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/csharp/csharp-generic-collections&usg=ALkJrhhhHoyQP35WLZUId6lYFxhvR4gQsg) namespaces como List <T>, Dictionary <T>, SortedList <T>, Queue <T> , HashSet <T>, LinkedList <T> etc.

La siguiente figura ilustra que los métodos de extensión incluidos en la clase Enumerable se pueden usar con la colección genérica en C # o VB.Net.

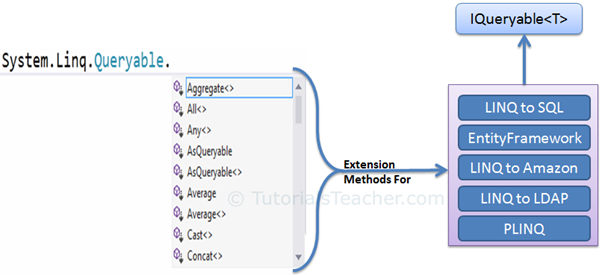
[](http://www.tutorialsteacher.com/Content/images/linq/Enumerable-extension-methods.png)Métodos de extensión IEnumerable <T> en la clase Enumerable

## Queryable:

La clase **Queryable** incluye métodos de extensión para las clases que implementan la interfaz [IQueryable <t>](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://msdn.microsoft.com/en-us/library/vstudio/bb351562(v%3Dvs.100).aspx&usg=ALkJrhh6aC88iGsiaUhQX4BSm2rhp3d13g) . IQueryable <T> se utiliza para proporcionar capacidades de consulta contra una fuente de datos específica donde se conoce el tipo de datos. Por ejemplo, Entity Framework api implementa la interfaz IQueryable <T> para admitir consultas LINQ con una base de datos subyacente como SQL Server.

Además, hay API disponibles para acceder a datos de terceros; por ejemplo, LINQ to Amazon ofrece la posibilidad de utilizar LINQ con los servicios web de Amazon para buscar libros y otros elementos mediante la implementación de la interfaz IQueryable.

La siguiente figura ilustra que los métodos de extensión incluidos en la clase Queryable se pueden usar con diversos proveedores de datos nativos o de terceros.

[](http://www.tutorialsteacher.com/Content/images/linq/Queryable-extension-methods.png)Métodos de extensión IQueryable en clase Queryable

Visite MSDN para conocer todos los métodos de extensión de la clase [Enumerable](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=https://msdn.microsoft.com/en-us/library/system.linq.enumerable(v%3Dvs.110).aspx&usg=ALkJrhgmn6vepFM6u3mGyQLVYb-nqjef6g) y [Queryable](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=https://msdn.microsoft.com/en-us/library/system.linq.queryable(v%3Dvs.110).aspx&usg=ALkJrhij9xQJ5eXvPOsI5Tl4CS9knjbuDA) .



### Puntos para recordar :

1. Use el espacio de nombres **System.Linq** para usar LINQ.
2. La API de LINQ incluye dos clases estáticas principales Enumerable y Queryable.
3. La clase **Enumerable** estática incluye métodos de extensión para las clases que implementan la interfaz IEnumerable <T>.
4. El tipo de colecciones IEnumerable <T> son colecciones en la memoria como List, Dictionary, SortedList, Queue, HashSet, LinkedList
5. La clase **Queryable** estática incluye métodos de extensión para las clases que implementan la interfaz IQueryable <T>
6. El proveedor de consultas remotas implementa IQueryable <T>. p.ej.Linq-to-SQL, LINQ-to-Amazon, etc.

Obtenga más información acerca de la sintaxis de consulta de LINQ.

# Sintaxis de consulta LINQ

Hay dos formas básicas de escribir una consulta LINQ en la colección IEnumerable o fuentes de datos IQueryable.

1. Query Sintaxis o Query Expression Syntax
2. Sintaxis del método o sintaxis de extensión del método o Fluido

## Sintaxis de consulta:

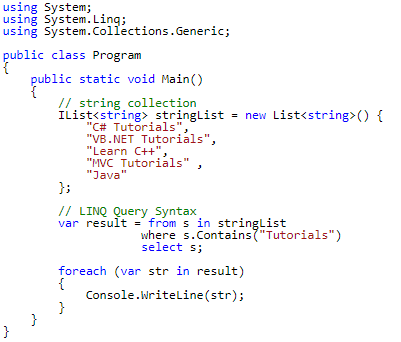
La sintaxis de consulta es similar a SQL (lenguaje de consulta estructurado) para la base de datos.Se define dentro del código C # o VB.

Sintaxis de consulta LINQ:

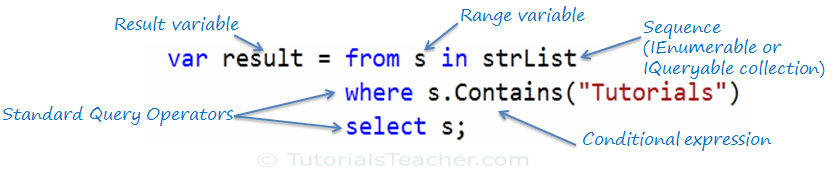
desde *<variable de rango>* en *<IEnumerable <T> o IQueryable <T> Collection>* <Operadores de consulta estándar> *<lambda expression>* <select or groupBy operator> *<result formation>*

La sintaxis de la consulta LINQ comienza con la palabra clave y finaliza con la palabra clave seleccionada. La siguiente es una consulta de muestra LINQ que devuelve una colección de cadenas que contiene una palabra "Tutoriales".

Ejemplo: sintaxis de consulta LINQ en C #



La siguiente figura muestra la estructura de la sintaxis de la consulta LINQ.

[](http://www.tutorialsteacher.com/Content/images/linq/linq-query-syntax.png)Sintaxis de consulta LINQ

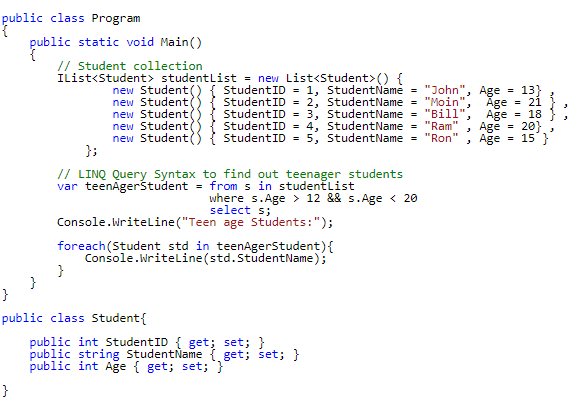
La sintaxis de consulta comienza con una cláusula ***From*** seguida de una variable ***Range*** . La cláusula ***From*** está estructurada como " **From** rangeV *ariableName* **in** *IEnumerablecollection* " . En inglés, esto significa, de cada objeto en la colección. Es similar a un bucle foreach(Student s in studentList) : foreach(Student s in studentList) .

Después de la cláusula De, puede usar diferentes Operadores de consultas estándar para filtrar, agrupar y unir elementos de la colección. Hay alrededor de 50 Operadores de consultas estándar disponibles en LINQ. En la figura anterior, hemos utilizado el operador "where" (cláusula aka) seguido de una condición. Esta condición generalmente se expresa usando la [expresión lambda](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/linq/linq-lambda-expression&usg=ALkJrhjSONdFXdmGfI9Lv4jWDSq4OPLAPA) .

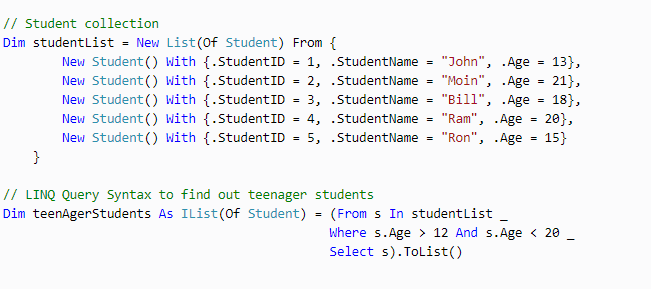
La sintaxis de consulta LINQ siempre termina con una cláusula Select o Group. La cláusula Select se usa para dar forma a los datos. Puede seleccionar el objeto completo tal como es o solo algunas propiedades del mismo. En el ejemplo anterior, seleccionamos los elementos de cuerda resultantes.

En el siguiente ejemplo, usamos la sintaxis de la consulta LINQ para encontrar estudiantes adolescentes de la colección de Estudiantes (secuencia).

Ejemplo: sintaxis de consulta LINQ en C #



Ejemplo: sintaxis de consulta LINQ en VB.Net





### Puntos para recordar :

1. Como su nombre lo sugiere, la **sintaxis de consulta** es similar a la **sintaxis de** SQL (lenguaje de consulta de estructura).
2. La sintaxis de consulta comienza con la cláusula *from* y puede finalizar con la cláusula *Select* o *GroupBy* .
3. Utilice varios otros operadores como filtrar, unir, agrupar, ordenar operadores para construir el resultado deseado.
4. [Variable implícitamente tipada - var](http://www.tutorialsteacher.com/csharp/csharp-var-implicit-typed-local-variable) se puede usar para contener el resultado de la consulta LINQ.

Aprende sobre la sintaxis del método a continuación.

# Sintaxis del método LINQ

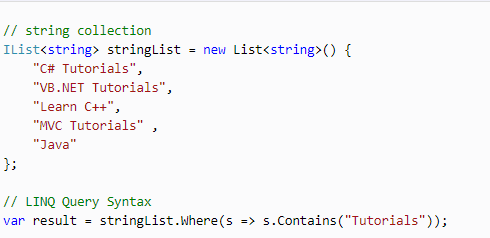
En la sección anterior, ha aprendido sobre la sintaxis de consulta LINQ. Aquí, aprenderá sobre la sintaxis del Método.

http://www.tutorialsteacher.com/Content/images/tips.pngEl compilador convierte la sintaxis de la consulta en sintaxis de método en tiempo de compilación.

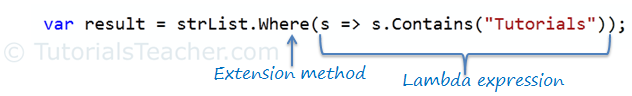
La sintaxis de los métodos (también conocida como sintaxis fluida) utiliza métodos de extensión incluidos en la clase estática **Enumerable** o **Queryable** , similar a cómo llamarías al método de extensión de cualquier clase.

La siguiente es una consulta de sintaxis del método LINQ de ejemplo que devuelve una colección de cadenas que contiene una palabra "Tutoriales".

Ejemplo: sintaxis del método LINQ en C #

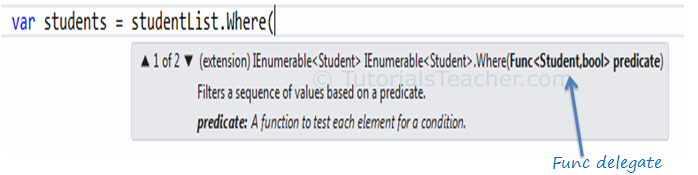


La siguiente figura ilustra la estructura de la sintaxis del método LINQ.

[](http://www.tutorialsteacher.com/Content/images/linq/linq-method-syntax.png)Estructura de sintaxis del método LINQ

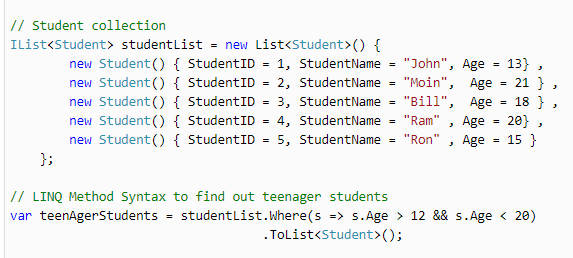
Como puede ver en la figura anterior, la sintaxis del método se compone de métodos de extensión y expresión Lambda. El método de extensión **Where ()** se define en la clase Enumerable.

Si comprueba la firma del método de extensión Where, encontrará que el método Where acepta un delegado [predicado](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/csharp/csharp-predicate&usg=ALkJrhh88wE_syJN1oX7aM7Bnp-lkV0fGQ) como Func <Student, bool>. Esto significa que puede pasar cualquier función delegada que acepte un objeto Student como parámetro de entrada y devuelve un valor booleano como se muestra en la figura siguiente. La expresión lambda funciona como un delegado pasado en la cláusula Where. Aprenda la expresión lambda en la siguiente sección.

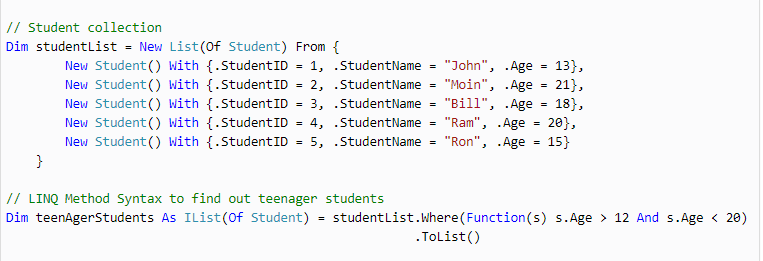
[](http://www.tutorialsteacher.com/Content/images/linq/linq-where-extension-method.png)Func delegate en Donde

El siguiente ejemplo muestra cómo usar la consulta de sintaxis del método LINQ con la colección IEnumerable <T>.

Ejemplo: sintaxis de método en C #



Ejemplo: sintaxis de método en VB.Net





### Puntos para recordar :

1. Como su nombre lo sugiere, **Method Syntax** es como llamar al método de extensión.
2. La **sintaxis del método** LINQ también conocida como sintaxis fluida porque permite la llamada a series de métodos de extensión.
3. Variable implícitamente tipada - var se puede usar para contener el resultado de la consulta LINQ.

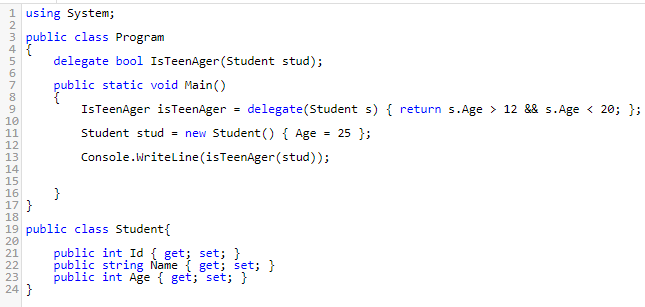
Aprenda sobre la expresión lambda siguiente.

# Anatomía de la expresión lambda

C # 3.0 (.NET 3.5) introdujo la expresión lambda junto con LINQ. La expresión lambda es una forma más corta de representar el [método anónimo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/csharp/csharp-anonymous-method&usg=ALkJrhhvv7r9HR0L8XPm3ZKkWWnI5HRtSw) utilizando alguna sintaxis especial.

Por ejemplo, siguiendo el método anónimo comprueba si el alumno es adolescente o no:

Método anónimo en C #:





Método anónimo en VB.Net:

Dim isStudentTeenAger = Function (s As Student ) As Boolean Return s.Age > 12 And s.Age < 20 End Function

Dim isStudentTeenAger = Function (s As Student ) As Boolean Return s.Age > 12 And s.Age < 20 End Function

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-77xAMs&usg=ALkJrhjjNPAYkBITe-b5selOpLJ3UKIBUg)

El método anónimo anterior se puede representar utilizando una expresión Lambda en C # y VB.Net como se muestra a continuación:

Expresión Lambda en C #:

s => s.Age > 12 && s.Age < 20

s => s.Age > 12 && s.Age < 20

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-rqSYnO&usg=ALkJrhj8JAxVQehXDbB1XHspZho471Px3A)

Lambda Expression en VB.Net:

Function (s) s.Age > 12 And s.Age < 20

Function (s) s.Age > 12 And s.Age < 20

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-R13yq1&usg=ALkJrhh1nsrMKaplfjTAaIqr4i8No6fa4g)

Veamos cómo evolucionó la expresión lambda a partir del siguiente método anónimo.

Método anónimo en C #:

delegate ( Student s) { return s.Age > 12 && s.Age < 20; };

delegate ( Student s) { return s.Age > 12 && s.Age < 20; };

La expresión Lambda se desarrolla a partir del método anónimo eliminando primero la palabra clave del delegado y el tipo de parámetro y agregando un operador lambda =>.

Expresión lambda del método anónimo

La expresión lambda anterior es absolutamente válida, pero no necesitamos las llaves, retorno y punto y coma si solo tenemos una declaración que devuelve un valor. Entonces podemos eliminarlo.

Además, podemos eliminar parenthesis (), si solo tenemos un parámetro.

Expresión lambda del método anónimo

Por lo tanto, obtuvimos la expresión lambda: s => s.Age > 12 && s.Age < 20 donde **s** es un parámetro, **=>** es el operador lambda y **s.Age > 12 && s.Age < 20** es la expresión corporal :

Estructura de expresión Lambda en C #

De la misma manera que obtuvimos la expresión lambda en VB.Net se puede escribir de la siguiente manera:

Estructura de expresión Lambda en VB.Net

La expresión lambda se puede invocar de la misma manera que delegate using ().

**Nota:VB.Net no es compatible con el operador lambda =>**

## Expresión lambda con múltiples parámetros:

Puede ajustar los parámetros entre paréntesis si necesita pasar más de un parámetro, como se muestra a continuación:

Ejemplo: especificar múltiples parámetros en la expresión lambda C #

**(s, youngAge)** => s.Age >= youngage;

**(s, youngAge)** => s.Age >= youngage;

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-PWGyET)

También puede dar el tipo de cada parámetro si los parámetros son confusos:

Ejemplo: especifique el tipo de parámetro en la expresión lambda C #

**( Student s, int youngAge)** => s.Age >= youngage;

**( Student s, int youngAge)** => s.Age >= youngage;

Ejemplo: especificar múltiples parámetros en la expresión lambda VB.Net

Function (s, youngAge) s.Age >= youngAge

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-52ROJw)

## Expresión Lambd sin ningún parámetro:

No es necesario tener al menos un parámetro en una expresión lambda. La expresión lambda se puede especificar sin ningún parámetro también.

Ejemplo: expresión Lambda con cero parámetro.

**()** => Console .WriteLine( "Parameter less lambda expression" )

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-1JQVyo)

## Múltiples declaraciones en la expresión corporal:

Puede ajustar expresiones con llaves si desea tener más de una declaración en el cuerpo:

Ejemplo: expresión Lambda C #

(s, youngAge) => **{** Console .WriteLine( "Lambda expression with multiple statements in the body" ); Return s.Age >= youngAge; **}**

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-oGZEjZ)

Ejemplo: expresión Lambda VB.Net

Function (s , youngAge) Console .WriteLine( "Lambda expression with multiple statements in the body" ) Return s.Age >= youngAge End Function

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-HA4Mqu)

## Variable local en Lambda Expression body:

Puede declarar una variable en el cuerpo de la expresión para usarla en cualquier parte del cuerpo de la expresión, como se muestra a continuación:

Ejemplo: expresión Lambda C #

s => { **int youngAge = 18;** Console .WriteLine( "Lambda expression with multiple statements in the body" ); return s.Age >= youngAge; }

s => { **int youngAge = 18;** Console .WriteLine( "Lambda expression with multiple statements in the body" ); return s.Age >= youngAge; }

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-S1K7xU)

Ejemplo: expresión Lambda VB.Net

Function (s) **Dim youngAge As Integer = 18** Console.WriteLine( "Lambda expression with multiple statements in the body" ) Return s.Age >= youngAge End Function

Function (s) **Dim youngAge As Integer = 18** Console.WriteLine( "Lambda expression with multiple statements in the body" ) Return s.Age >= youngAge End Function

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-p76aBt)

La expresión Lambda también se puede asignar a delegados incorporados, como [Func](http://www.tutorialsteacher.com/csharp/csharp-func-delegate) , [Action](http://www.tutorialsteacher.com/csharp/csharp-action-delegate) y [Predicate](http://www.tutorialsteacher.com/csharp/csharp-predicate) .

## Delegado Func:

Utilice el delegado Func <> cuando desee devolver algo desde una expresión lambda. El último tipo de parámetro en un delegado Func <> es el tipo de retorno y el resto son parámetros de entrada.Visite la sección de [delegados de Func](http://www.tutorialsteacher.com/csharp/csharp-func-delegate) de tutoriales de C # para obtener más información al respecto.

Considere la siguiente expresión lambda para saber si un alumno es un adolescente o no.

Ejemplo: expresión Lambda asignada al delegado Func en C #

Func < Student , bool > isStudentTeenAger = s => s.age > 12 && s.age < 20; Student std = new Student () { age = 21 }; bool isTeen = isStudentTeenAger(std); // returns false

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-MfwAY6)

Ejemplo: expresión de Lamda asignada al delegado de Func en VB.Net

Dim isStudentTeenAger As Func (Of Student , Boolean ) = Function (s) s.Age > 12 And s.Age < 20 Dim stud As New Student With {.Age = 21} Dim isTeen As Boolean = isStudentTeenAger(stud) // returns false

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-dh5cUY)

En el ejemplo anterior, el delegado Func espera que el primer parámetro de entrada sea de tipo Alumno y el tipo de devolución sea booleano. La expresión lambda s => s.age > 12 && s.age < 20satisface el requisito de delegado Func <Student, bool>, como se muestra a continuación:

Func delegate con Lambda Expression

El delegado Func <> que se muestra arriba, se convertiría en una función como se muestra a continuación.

DO#:

bool isStudentTeenAger( Student s) { return s.Age > 12 && s.Age < 20; }

## Delegado de Acción:

A diferencia del delegado de Func, un delegado de Acción solo puede tener parámetros de entrada.Use el tipo de [delegado Acción](http://www.tutorialsteacher.com/csharp/csharp-action-delegate) cuando no necesite devolver ningún valor de la expresión lambda.

Ejemplo: expresión de Lamda asignada al delegado de Acción en C #

Action < Student > PrintStudentDetail = s => Console .WriteLine( "Name: {0}, Age: {1} " , s.StudentName, s.Age); Student std = new Student (){ StudentName = "Bill" , Age=21}; PrintStudentDetail(std); //output: Name: Bill, Age: 21

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-TilB9f)

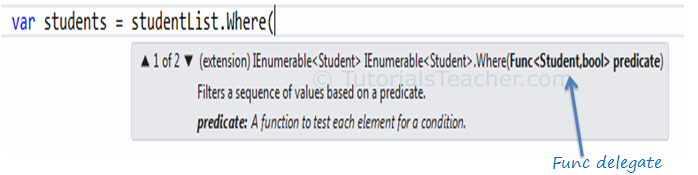
Ejemplo: expresión de Lamda asignada al delegado de Acción en VB.Net

Dim printStudentDetail As Action(Of Student) = Sub (s) Console.WriteLine( "Name: {0}, Age: {1} " , s.StudentName, s.Age) Dim stud As New Student With {.StudentName = "Bill" , .Age = 21} printStudentDetail(stud) //output: Name: Bill, Age: 21

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-nCWoaG)

## Expresión lambda en consulta LINQ:

Por lo general, la expresión lambda se usa con la consulta LINQ. La clase estática IEnumerable<T>incluye el método de extensión Where para IEnumerable<T> que acepta Func<TSource,bool> . Por lo tanto, se requiere el método de extensión Where () para la IEnumerable<Student> para pasar Func<Student,bool> , como se muestra a continuación:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-where-extension-method.png)Parámetro de delegado de Func en el método de extensión Where

Entonces, ahora puede pasar la expresión lambda asignada al delegado Func al método de extensión Where () en la sintaxis del método como se muestra a continuación:

Ejemplo: delegado de Func en la sintaxis del método LINQ

IList < Student > studentList = new List < Student >(){...}; Func < Student , bool > isStudentTeenAger = s => s.age > 12 && s.age < 20; var teenStudents = studentList.Where(isStudentTeenAger). ToList < Student >();

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-tCgBt7)

Ejemplo: delegado de Func en la sintaxis de consulta de LINQ

IList < Student > studentList = new List < Student >(){...}; Func < Student , bool > isStudentTeenAger = s => s.age > 12 && s.age < 20; var teenStudents = from s in studentList where isStudentTeenAger(s) select s;

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-VmWuAS)

Puede seguir el mismo método en VB.Net para pasar el delegado de Func.



### Puntos para recordar:

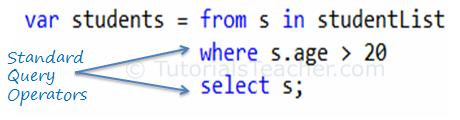
1. Lambda Expression es una forma más corta de representar el método anónimo.
2. Sintaxis de la expresión Lambda: *parameters => body expression*
3. Lambda Expression puede tener cero parámetro.
4. Lambda Expression puede tener múltiples parámetros entre paréntesis ().
5. Lambda Expression puede tener múltiples instrucciones en la expresión corporal entre llaves {}.
6. Lambda Expression se puede asignar a Func, Action o Predicate delegate.
7. Lambda Expression se puede invocar de forma similar a delegar.

Obtenga más información sobre los Operadores de consultas estándar en el próximo capítulo.

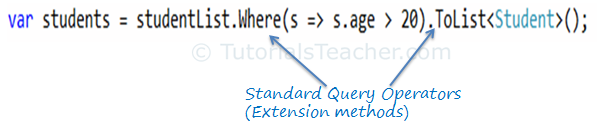
# Operadores de consulta estándar:

Los Operadores de consultas estándar en LINQ son en realidad métodos de extensión para los IEnumerable<T> and IQueryable<T> . Se definen en las clases System.Linq.Enumerable y System.Linq.Queryable . Hay más de 50 operadores de consulta estándar disponibles en LINQ que proporcionan diferentes funcionalidades como filtrado, clasificación, agrupación, agregación, concatenación, etc.

## Operadores de consulta estándar en la sintaxis de consulta:

[](http://www.tutorialsteacher.com/Content/images/linq/standard-query-operators-linq-query-syntax.png)Operadores de consultas estándar en la sintaxis de consultas

## Operadores de consultas estándar en la sintaxis del método:

[](http://www.tutorialsteacher.com/Content/images/linq/standard-query-operators-linq-method-syntax.png)Operadores de consultas estándar en la sintaxis de métodos

Los operadores de consulta estándar en sintaxis de consulta se convierten en métodos de extensión en tiempo de compilación. Entonces ambos son lo mismo.

Los operadores de consultas estándar se pueden clasificar en función de la funcionalidad que proporcionan. La siguiente tabla enumera toda la clasificación de Operadores de consultas estándar:

| **Clasificación** | **Operadores de consultas estándar** |
| --- | --- |
| Filtración | Donde, OfType |
| Clasificación | OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse |
| Agrupamiento | GroupBy, ToLookup |
| Unirse | GroupJoin, Únete |
| Proyección | Seleccionar, SeleccionarMuchas |
| Agregación | Agregado, Promedio, Cuenta, LongCount, Máx., Mín., Suma |
| Cuantificadores | Todo, cualquiera, contiene |
| Elementos | ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault |
| Conjunto | Distinto, Excepto, Intersecar, Unión |
| Particionamiento | Skip, SkipWhile, Take, TakeWhile |
| Concatenación | Concat |
| Igualdad | SequenceEqual |
| Generacion | DefaultEmpty, Empty, Range, Repeat |
| Conversión | AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList |

Aprende cada operador de consulta estándar en las siguientes secciones.

# Operadores de filtrado - Donde

Los operadores de filtrado en LINQ filtran la secuencia (recopilación) en función de algunos criterios dados.

La siguiente tabla enumera todos los operadores de filtrado disponibles en LINQ.

| **Operadores de filtrado** | **Descripción** |
| --- | --- |
| Dónde | Devuelve valores de la colección en función de una función de predicado |
| [OfType](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/linq/linq-filtering-operators-oftype&usg=ALkJrhiLSjsl14MHvQihU78VWHpOlMYAzg) | Devuelve valores de la colección en función de un tipo especificado. Sin embargo, dependerá de su capacidad para convertir a un tipo específico. |

## Dónde

El operador Where (método de extensión Linq) filtra la colección en función de un criterio determinado. Acepta un [predicado](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/csharp/csharp-predicate&usg=ALkJrhh88wE_syJN1oX7aM7Bnp-lkV0fGQ) como un parámetro.

El método de extensión **Where** tiene dos sobrecargas posteriores. Una sobrecarga requiere el parámetro de entrada Func <TSource, bool> y el segundo método de sobrecarga Func <TSource, int, bool> parámetro de entrada donde int es para el índice:

public static IEnumerable <TSource> Where<TSource>( this IEnumerable <TSource> source, Func <TSource, bool > predicate); public static IEnumerable <TSource> Where<TSource>( this IEnumerable <TSource> source, Func <TSource, int , bool > predicate);

### Donde cláusula en Query Sintaxis:

La siguiente muestra de consulta utiliza un operador Where para filtrar a los estudiantes que son adolescentes de la colección dada (secuencia). Utiliza una expresión lambda como función predicativa.

Ejemplo: cláusula Where: sintaxis de consulta LINQ en C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var filteredResult = from s in studentList where s.Age > 12 && s.Age < 20 select s.StudentName;

Ejemplo: cláusula Where: sintaxis de consulta LINQ en VB.Net

Dim studentList = Nueva lista (de estudiante ) de {

New Student () con {.StudentID = 1, .StudentName = "John" , .Age = 18},

New Student () con {.StudentID = 2, .StudentName = "Steve" , .Age = 15},

New Student () con {.StudentID = 3, .StudentName = "Bill" , .Age = 25},

New Student () con {.StudentID = 4, .StudentName = "Ram" , .Age = 20},

Nuevo estudiante () con {.StudentID = 5, .StudentName = "Ron" , .Age = 19}

}

Dim filteredResult = From s En studentList

Donde a la edad> 12 y a la edad <20

Seleccione s.StudentName

En el ejemplo anterior, filtradoResult incluirá los siguientes estudiantes después de la ejecución de la consulta.

John  
Steve  
Ron

En la consulta de muestra anterior, el cuerpo de la expresión lambda en la **s.Age > 12 && s.Age < 20**se pasa como una función de predicado **Func <TSource, bool >** que evalúa a cada alumno de la colección.

Alternativamente, también puede usar un delegado de tipo Func con un método anónimo para pasar como una función de predicado como a continuación (la salida sería la misma):

Exapmle: Where cláusula

Func < Student , bool > isTeenAger = delegate ( Student s) { return s.Age > 12 && s.Age < 20; }; var filteredResult = from s in studentList where isTeenAger(s) select s;

### Donde el método de extensión en la sintaxis del método:

A diferencia de la sintaxis de la consulta, debe pasar la expresión lambda completa como una función de predicado en lugar de solo la expresión corporal en la sintaxis del método LINQ.

Ejemplo: Donde en la sintaxis del método en C #

var filteredResult = studentList.Where(s => s.Age > 12 && s.Age < 20);

Ejemplo: Donde en la sintaxis del método en VB.Net

Dim filteredResult = studentList.Where( Function (s) s.Age > 12 And s.Age < 20 )

Como se mencionó anteriormente, el método de extensión **Where** también tiene una segunda sobrecarga que incluye el índice del elemento actual en la colección. Puede usar ese índice en su lógica si lo necesita.

El siguiente ejemplo usa la cláusula Where para filtrar los elementos impares en la colección y devolver solo los elementos pares. Recuerde que el índice comienza desde cero.

Ejemplo: Linq - Donde el método de extensión en C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var filteredResult = studentList.Where((s, i) => { if (i % 2 == 0) // if it is even element return true ; return false ; }); foreach ( var std in filteredResult) Console .WriteLine(std.StudentName);

Salida:

John  
Cuenta  
Ron

### Cláusula Multiple Where:

Puede llamar al método de extensión Where () más de una vez en una única consulta LINQ.

Ejemplo: cláusula Multiple where en Query Syntax C #

var filteredResult = from s in studentList where s.Age > 12 where s.Age < 20 select s;

Ejemplo: cláusula Multiple where en Method Syntax C #

var filteredResult = studentList.Where(s => s.Age > 12).Where(s => s.Age < 20);



### Puntos para recordar :

1. **Donde** se usa para filtrar la colección en base a los criterios dados.
2. Donde el método de extensión tiene dos métodos de sobrecarga. Use un segundo método de sobrecarga para conocer el índice del elemento actual en la colección.
3. La sintaxis del método requiere toda la expresión lambda en el método de extensión Where mientras que la sintaxis de consulta solo requiere cuerpo de expresión.
4. Múltiple **Donde los** métodos de extensión son válidos en una sola consulta LINQ.

Aprenda otro operador de filtrado - OfType en la siguiente sección.

# Operador de filtrado - OfType

El operador OfType filtra la colección en función de la capacidad de convertir un elemento en una colección en un tipo especificado.

## OfType in Query Syntax:

Utilice el operador OfType para filtrar la colección anterior en función del tipo de elemento

Ejemplo: operador OfType en C #

IList mixedList = new ArrayList (); mixedList.Add(0); mixedList.Add( "One" ); mixedList.Add( "Two" ); mixedList.Add(3); mixedList.Add( new Student() { StudentID = 1, StudentName = "Bill" }); var stringResult = from s in mixedList.OfType< string >() select s; var intResult = from s in mixedList.OfType< int >() select s;

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-VVUIZs)

Ejemplo: Operador OfType en VB.Net:

Dim stringResult = From s In mixedList.OfType(Of String )()

Las consultas de ejemplo anteriores devolverán elementos cuyo tipo es una cadena en mixedList.stringResult contiene los siguientes elementos después de la ejecución:

Uno  
Dos  
0  
3  
Cuenta

## OfType in Method Sintaxis:

Puede utilizar el método de extensión OfType <TResult> () en la sintaxis del método linq como se muestra a continuación.

Ejemplo: OfType en C #

var stringResult = mixedList.OfType< string >();

Ejemplo: OfType en VB.Net

Dim stringResult = mixedList.OfType(Of String )

Dim stringResult = mixedList.OfType(Of String )

stringResult contendría los siguientes elementos.

Uno  
Dos



### Puntos para recordar :

1. El operador **Where** filtra la recopilación en función de una función de predicado.
2. El operador **OfType** filtra la colección en función de un tipo determinado
3. **Los** métodos de extensión **Where** y **OfType** pueden **invocarse** varias veces en una sola consulta LINQ.

Visite MSDN para obtener más información sobre los [operadores de filtrado](http://msdn.microsoft.com/en-us/library/bb546161.aspx) .

# Operadores de clasificación: OrderBy & OrderByDescending

Un operador de clasificación organiza los elementos de la colección en orden ascendente o descendente. LINQ incluye los siguientes operadores de clasificación.

| **Operador de clasificación** | **Descripción** |
| --- | --- |
| OrderBy | Ordena los elementos en la colección en función de los campos especificados en orden ascendente o descendente. |
| OrderByDescending | Ordena la colección según los campos especificados en orden descendente. Solo es válido en la sintaxis del método. |
| Entonces por | Solo es válido en la sintaxis del método. Se utiliza para la clasificación de segundo nivel en orden ascendente. |
| ThenByDescending | Solo es válido en la sintaxis del método. Se usa para la clasificación de segundo nivel en orden descendente. |
| Marcha atrás | Solo es válido en la sintaxis del método. Ordena la colección en orden inverso. |

# OrderBy:

OrderBy ordena los valores de una colección en orden ascendente o descendente. Clasifica la colección en orden ascendente de manera predeterminada porque la palabra clave ascending es opcional aquí. Use la palabra clave descendente para ordenar la colección en orden descendente.

Ejemplo: OrderBy en la sintaxis de consulta C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var orderByResult = from s in studentList orderby s.StudentName select s; var orderByDescendingResult = from s in studentList orderby s.StudentName descending select s;

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-f5ywFg&usg=ALkJrhgzyTBxUmEXRWnlrPgzzTjNRDBPBw)

Ejemplo: OrderBy en la sintaxis de consulta VB.Net

Dim orderByResult = From s In studentList Order By s.StudentName Select s Dim orderByDescendingResult = From s In studentList Order By s.StudentName Descending Select s

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-Zef7G3&usg=ALkJrhgCAtkFfBAecNSo7OFtr3gu1qeLVA)

orderByResult en el ejemplo anterior contendría los siguientes elementos después de la ejecución:

Cuenta  
John  
RAM  
Ron  
Steve

orderByDescendingResult en el ejemplo anterior contendría los siguientes elementos después de la ejecución:

Steve  
Ron  
RAM  
John  
Cuenta

## OrderBy en la sintaxis del método:

El método de extensión OrderBy tiene dos sobrecargas. La primera sobrecarga del método de extensión OrderBy acepta el parámetro de tipo de delegado Func. Por lo tanto, debe pasar la expresión lambda para el campo en función del cual desea ordenar la colección.

El segundo método de sobrecarga de OrderBy acepta el objeto de IComparer junto con el tipo de delegado Func para usar una comparación personalizada para la ordenación.

Sobrecargas del método OrderBy ():

public static IOrderedEnumerable <TSource> OrderBy <TSource, TKey> ( esta fuente IEnumerable <TSource>,

Func <TSource, TKey> keySelector);

public static IOrderedEnumerable <TSource> OrderBy <TSource, TKey> ( esta fuente IEnumerable <TSource>,

Func <TSource, TKey> keySelector,

Comparador IComparer <TKey>);

El siguiente ejemplo ordena la colección studentList en orden ascendente de StudentName utilizando el método de extensión OrderBy.

Ejemplo: OrderBy en la sintaxis del método C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var studentsInAscOrder = studentList.OrderBy(s => s.StudentName);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-zJ5SmO&usg=ALkJrhgeangQ3_wTFp7_GadskPw5GcQxmg)

Ejemplo: OrderBy en la sintaxis del método VB.Net

Dim studentsInAscOrder = studentList.OrderBy ( Function (s) s.StudentName)

**Nota: Lasintaxis del método no permite que la palabra clave decending ordene la colección en orden descendente.Use el método OrderByDecending () para ello.**

## OrderByDescending:

OrderByDescending ordena la colección en orden descendente.

OrderByDescending es válido solo con la sintaxis Method. No es válido en la sintaxis de la consulta porque la sintaxis de la consulta utiliza atributos ascendentes y descendentes como se muestra arriba.

Ejemplo: OrderByDescending C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var studentsInDescOrder = studentList.OrderByDescending(s => s.StudentName);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-zJ5SmO&usg=ALkJrhgeangQ3_wTFp7_GadskPw5GcQxmg)

Ejemplo: OrderByDescending VB.Net

Dim studentsInDescOrder = studentList.OrderByDescending( Function (s) s.StudentName)

Un resultado en el ejemplo anterior contendría los siguientes elementos después de la ejecución.

Steve  
Ron  
RAM  
John  
Cuenta

Tenga en cuenta que OrderByDescending no es compatible con la sintaxis de la consulta. Use la palabra clave decending en su lugar.

## Clasificación múltiple:

Puede ordenar la colección en múltiples campos separados por coma. La colección dada se ordenaría primero según el primer campo y luego, si el valor del primer campo sería el mismo para dos elementos, usaría el segundo campo para ordenar, y así sucesivamente.

Ejemplo: clasificación múltiple en la sintaxis de consulta C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 }, new Student () { StudentID = 6, StudentName = "Ram" , Age = 18 } }; var orderByResult = from s in studentList orderby s.StudentName, s.Age select new { s.StudentName, s.Age };

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-0KqpNz&usg=ALkJrhio8vjPJIRtowqgLgHXt98KGW00Iw)

En el ejemplo anterior, la colección studentList incluye dos StudentNames idénticos, Ram. Así que ahora, studentList se ordenaría primero en base a StudentName y luego por Age en orden ascendente. Entonces, orderByResult contendría los siguientes elementos después de la ejecución

StudentName: Bill, Edad: 25  
StudentName: John, Edad: 18  
Nombre del alumno: Ram, Edad: 18  
Nombre del estudiante: Ram, Edad: 20  
Nombre del estudiante: Ron, Edad: 19  
Nombre del estudiante: Steve, Edad: 15

**Nota:la clasificación múltiple en la sintaxis del método funciona de manera diferente.Utilice los métodos de extensión ThenBy o ThenByDecending para la clasificación secundaria.**



### Puntos para recordar:

1. LINQ incluye cinco operadores de clasificación: OrderBy, OrderByDescending, ThenBy, ThenByDescending y Reverse
2. La sintaxis de consulta de LINQ no es compatible con OrderByDescending, ThenBy, ThenByDescending y Reverse. Solo admite la cláusula 'Order By' con una dirección de clasificación 'ascendente' y 'descendente'.
3. La sintaxis de la consulta LINQ admite varios campos de clasificación separados por coma, mientras que debe utilizar los métodos ThenBy y ThenByDescending para la clasificación secundaria.

Conozca los métodos ThenBy () y ThenByDescending () en la próxima sección.

# Operadores de clasificación: ThenBy & ThenByDescending

Hemos visto cómo hacer la clasificación usando múltiples campos en la sintaxis de la consulta en la sección anterior.

La ordenación múltiple en la sintaxis del método es compatible con los métodos de extensión ThenBy y ThenByDescending.

El método OrderBy () ordena la colección en orden ascendente en función del campo especificado.Utilice el método ThenBy () después de OrderBy para ordenar la colección en otro campo en orden ascendente. Linq primero ordenará la colección en función del campo primario que se especifica mediante el método OrderBy y luego ordenará la colección resultante en orden ascendente de nuevo según el campo secundario especificado por el método ThenBy.

De la misma manera, utilice el método ThenByDescending para aplicar la ordenación secundaria en orden descendente.

El siguiente ejemplo muestra cómo usar el método ThenBy y ThenByDescending para la clasificación de segundo nivel:

Ejemplo: ThenBy & ThenByDescending

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 }, new Student () { StudentID = 6, StudentName = "Ram" , Age = 18 } }; var thenByResult = studentList.OrderBy(s => s.StudentName).ThenBy(s => s.Age); var thenByDescResult = studentList.OrderBy(s => s.StudentName).ThenByDescending(s => s.Age);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-0KqpNz&usg=ALkJrhio8vjPJIRtowqgLgHXt98KGW00Iw)

Como puede ver en el ejemplo anterior, primero ordenamos una colección studentList por StudentName y luego por Age. Así que ahora, entoncesByResult contendría elementos follwoing después de la clasificación:

StudentName: Bill, Edad: 25  
StudentName: John, Edad: 18  
Nombre del alumno: Ram, Edad: 18  
Nombre del estudiante: Ram, Edad: 20  
Nombre del estudiante: Ron, Edad: 19  
Nombre del estudiante: Steve, Edad: 15

thenByDescResult contendría los siguientes elementos. Tenga en cuenta que Ram con 20 años viene antes que Ram con 18 años porque ha utilizado ThenByDescending.

StudentName: Bill, Edad: 25  
StudentName: John, Edad: 18  
Nombre del estudiante: Ram, Edad: 20  
Nombre del alumno: Ram, Edad: 18  
Nombre del estudiante: Ron, Edad: 19  
Nombre del estudiante: Steve, Edad: 15

Puede usar el método ThenBy y ThenByDescending de la misma manera en VB.Net como se muestra a continuación:

Ejemplo: ThenBy & ThenByDescending VB.Net

Dim sortedResult = studentList.OrderBy( Function (s) s.StudentName) .ThenBy( Function (s) s.Age) Dim sortedResult = studentList.OrderBy( Function (s) s.StudentName) .ThenByDescending( Function (s) s.Age)

Dim sortedResult = studentList.OrderBy( Function (s) s.StudentName) .ThenBy( Function (s) s.Age) Dim sortedResult = studentList.OrderBy( Function (s) s.StudentName) .ThenByDescending( Function (s) s.Age)



### Puntos para recordar :

1. OrderBy y ThenBy ordenan las colecciones en orden ascendente de forma predeterminada.
2. ThenBy o ThenByDescending se usa para la clasificación de segundo nivel en la sintaxis del método.
3. El método ThenByDescending ordena la colección en orden descendente en otro campo.
4. ThenBy o ThenByDescending NO se aplica en la sintaxis de Query.
5. Aplique la clasificación secundaria en la sintaxis de la consulta separando los campos mediante la coma.

Más información sobre los operadores de agrupamiento a continuación.

# Agrupación de operadores: GroupBy & ToLookup

Los operadores de agrupación hacen lo mismo que la cláusula GroupBy de consulta SQL. Los operadores de agrupación crean un grupo de elementos basados ​​en la clave dada. Este grupo está contenido en un tipo especial de colección que implementa una interfaz IGrouping <TKey, TSource> donde TKey es un valor clave, en el que se ha formado el grupo y TSource es la colección de elementos que coincide con el valor de la clave de agrupación.

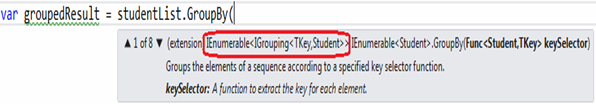
| **Operadores de Agrupación** | **Descripción** |
| --- | --- |
| Agrupar por | El operador GroupBy devuelve grupos de elementos en función de algún valor clave. Cada grupo está representado por el objeto IGrouping <TKey, TElement>. |
| Para buscar | ToLookup es lo mismo que GroupBy; la única diferencia es que la ejecución de GroupBy se aplaza mientras que la ejecución de ToLookup es inmediata. |

## Agrupar por:

http://www.tutorialsteacher.com/Content/images/tips.pngUna consulta LINQ puede terminar con una cláusula GroupBy o Select.

El operador GroupBy devuelve un grupo de elementos de la colección dada en función de algún valor clave. Cada grupo está representado por el objeto IGrouping <TKey, TElement>.Además, el método GroupBy tiene ocho métodos de sobrecarga, por lo que puede usar el método de extensión apropiado según su requisito en la sintaxis del método.

El resultado de los operadores de GroupBy es una colección de grupos. Por ejemplo, GroupBy devuelve IEnumerable <IGrouping <TKey, Student >> de la colección de estudiantes:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-groupby.png)Tipo de devolución de GroupBy ()

### GroupBy in Query Syntax:

El siguiente ejemplo crea un grupo de estudiantes que tienen la misma edad. Los estudiantes de la misma edad estarán en la misma colección y cada colección agrupada tendrá una colección clave e interior, donde la clave será la edad y la colección interna incluirá a los estudiantes cuya edad se corresponda con una clave.

Ejemplo: GroupBy in Query sintaxis C #

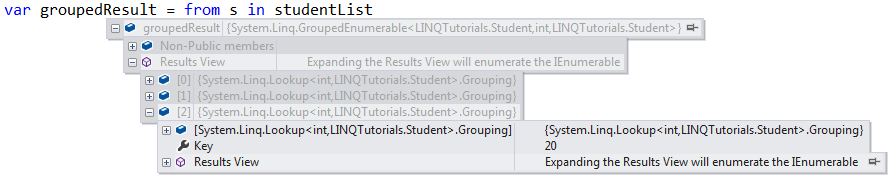
IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Abram" , Age = 21 } }; var groupedResult = from s in studentList group s by s.Age; //iterate each group foreach ( var ageGroup in groupedResult) { Console .WriteLine( "Age Group: {0}" , ageGroup .Key); //Each group has a key foreach ( Student s in ageGroup) // Each group has inner collection Console .WriteLine( "Student Name: {0}" , s.StudentName); }

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-BZnxR2&usg=ALkJrhgwOhSYb2PJw0BJCHzORXQnVtrA-A)

Salida:

AgeGroup: 18  
Nombre del estudiante: John  
Nombre del estudiante: Bill  
AgeGroup: 21  
Nombre del estudiante: Steve  
StudentName: Abram  
AgeGroup: 20  
StudentName: Ram

Como puede ver en el ejemplo anterior, puede iterar el grupo utilizando un bucle 'foreach', donde cada grupo contiene una clave y una colección interna. La siguiente figura muestra el resultado en la vista de depuración.

[](http://www.tutorialsteacher.com/Content/images/linq/linq-groupby-2.png)Colección agrupada con llave e interior de la colección

Use "Into Group" con la cláusula "Group By" en VB.Net como se muestra a continuación.

Ejemplo: cláusula GroupBy en VB.Net

Dim groupQuery = From s In studentList Group By s.Age Into Group For Each group In groupQuery Console .WriteLine( "Age Group: {0}" , group.Age) // Each group has key property name For Each student In group.Group // Each group has inner collection Console .WriteLine( "Student Name: {0}" , student.StudentName) Next Next

Dim groupQuery = From s In studentList Group By s.Age Into Group For Each group In groupQuery Console .WriteLine( "Age Group: {0}" , group.Age) // Each group has key property name For Each student In group.Group // Each group has inner collection Console .WriteLine( "Student Name: {0}" , student.StudentName) Next Next

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-cCMaRM&usg=ALkJrhhcG6iK2pvB63utTZbVZaccLHwS-Q)

Tenga en cuenta que cada grupo tendrá un nombre de propiedad sobre el grupo que se realizará.En el ejemplo anterior, hemos utilizado Age para formar un grupo, por lo que cada grupo tendrá el nombre de la propiedad "Edad" en lugar de "Clave" como nombre de propiedad.

Salida:

AgeGroup: 18  
Nombre del estudiante: John  
Nombre del estudiante: Bill  
AgeGroup: 21  
Nombre del estudiante: Steve  
StudentName: Abram  
AgeGroup: 20  
StudentName: Ram

### GroupBy in Method Syntax:

El método de extensión GroupBy () funciona de la misma manera en la sintaxis del método.Especifique la expresión lambda para el nombre de campo del selector de clave en el método de extensión GroupBy.

Ejemplo: GroupBy en la sintaxis del método C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Abram" , Age = 21 } }; var groupedResult = studentList.GroupBy(s => s.Age); foreach ( var ageGroup in groupedResult) { Console .WriteLine( "Age Group: {0}" , ageGroup.Key); //Each group has a key foreach (Student s in ageGroup) //Each group has a inner collection Console .WriteLine( "Student Name: {0}" , s.StudentName); }

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-9yg2ep&usg=ALkJrhg2lxhOdrj0OSG7XsdmDb0yk0WAPA)

Ejemplo: GroupBy en la sintaxis del método VB.Net

Dim groupQuery = studentList.GroupBy( Function (s) s.Age) For Each ageGroup In groupQuery Console .WriteLine( "Age Group: {0}" , ageGroup.Key) //Each group has a key For Each student In ageGroup.AsEnumerable() //Each group has a inner collection Console .WriteLine( "Student Name: {0}" , student.StudentName) Next Next

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-lNVo31&usg=ALkJrhhu4RKyI7STS3NHcixaP0OKJijwzw)

Salida:

AgeGroup: 18  
Nombre del estudiante: John  
Nombre del estudiante: Bill  
AgeGroup: 21  
Nombre del estudiante: Steve  
StudentName: Abram  
AgeGroup: 20  
StudentName: Ram

## Para buscar

ToLookup es lo mismo que GroupBy; la única diferencia es que la ejecución de GroupBy es diferida, mientras que la ejecución de ToLookup es inmediata. Además, ToLookup solo es aplicable en la sintaxis del Método. **ToLookup no es compatible con la sintaxis de la consulta.**

Ejemplo: Tocar en la sintaxis del método C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Abram" , Age = 21 } }; var lookupResult = studentList.ToLookup(s => s.age);

foreach (grupo var en lookupResult)

{

Consola .WriteLine ( "Grupo de edad: {0}" , group.Key); // Cada grupo tiene una clave

foreach (Estudiantes en grupo) // Cada grupo tiene una colección interna

Consola .WriteLine ( "Nombre del estudiante: {0}" , s.StudentName);

}

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-ZqU8cf&usg=ALkJrhgXBbgYGocDWVEEPZYfn6zdvHEhPw)

Ejemplo: Tocando en la sintaxis del método VB.Net

Dim loopupResult = studentList.ToLookup( Function (s) s.Age)

Dim loopupResult = studentList.ToLookup( Function (s) s.Age)



### Puntos para recordar :

1. GroupBy & ToLookup devuelven una colección que tiene una clave y una colección interna basada en un valor de campo clave.
2. La ejecución de GroupBy se difiere mientras que la de ToLookup es inmediata.
3. Una sintaxis de consulta LINQ puede finalizar con la cláusula GroupBy o Select.

Más información sobre Operador de unión.

# Operador que se une: Únete

Los operadores de unión se unen a las dos secuencias (colecciones) y producen un resultado.

| **Unirse a operadores** | **Uso** |
| --- | --- |
| Unirse | El operador de unión une dos secuencias (colecciones) basadas en una clave y devuelve una secuencia resultante. |
| GrupoÚnete | El operador GroupJoin une dos secuencias basadas en claves y devuelve grupos de secuencias. Es como Left External Join of SQL. |

## Unirse:

El operador de unión opera en dos colecciones, colección interna y colección externa. Devuelve una nueva colección que contiene elementos de ambas colecciones que satisfacen la expresión especificada. Es lo mismo que **la combinación interna** de SQL.

### Únase a la sintaxis del método:

El método de extensión de unión tiene dos sobrecargas, como se muestra a continuación.

Sobrecargas del método Join ():

public static IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> ( este IEnumerable <TOuter> externo,

IEnumerable <TInner> inner, Func <TOuter, TKey> outerKeySelector,

Func <TInner, TKey> innerKeySelector,

Func <TOuter, TInner, TResult> resultSelector);

public static IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> ( este IEnumerable <TOuter> externo,

IEnumerable <TInner> interno,

Func <TOuter, TKey> outerKeySelector,

Func <TInner, TKey> innerKeySelector,

Func <TOuter, TInner, TResult> resultSelector,

IEqualityComparer <TKey> comparador);

Como puede ver en el primer método de sobrecarga, toma cinco parámetros de entrada (excepto el primer parámetro 'this'): 1) externo 2) interno 3) outerKeySelector 4) innerKeySelector 5) resultSelector.

Tomemos un ejemplo simple. El siguiente ejemplo une dos colecciones de cadenas y devuelve una nueva colección que incluye cadenas coincidentes en ambas colecciones.

Ejemplo: Unir operador C #

IList < string > strList1 = new List < string >() { "One" , "Two" , "Three" , "Four" }; IList < string > strList2 = new List < string >() { "One" , "Two" , "Five" , "Six" }; var innerJoin = strList1.Join(strList2, str1 => str1, str2 => str2, (str1, str2) => str1);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-DoAglF&usg=ALkJrhiqgsJLCwg0EHxUuDlhCxnutgbj7w)

Uno  
Dos

Ahora, comprendamos join method usando la clase Student y Standard, donde la clase Student incluye StandardID que coincide con StandardID de la clase Standard.

Clases de ejemplo

public class Student { public int StudentID { get; set; } public string StudentName { get; set; } public int StandardID { get; set; } } public class Standard { public int StandardID { get; set; } public string StandardName { get; set; } }

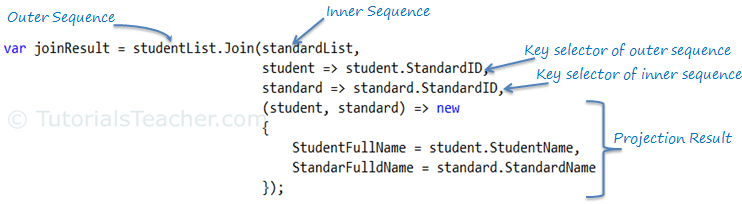
El siguiente ejemplo muestra la consulta LINQ Join.

Ejemplo: Join Query C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , StandardID =1 }, new Student () { StudentID = 2, StudentName = "Moin" , StandardID =1 }, new Student () { StudentID = 3, StudentName = "Bill" , StandardID =2 }, new Student () { StudentID = 4, StudentName = "Ram" , StandardID =2 }, new Student () { StudentID = 5, StudentName = "Ron" } }; IList < Standard > standardList = new List < Standard >() { new Standard (){ StandardID = 1, StandardName= "Standard 1" }, new Standard (){ StandardID = 2, StandardName= "Standard 2" }, new Standard (){ StandardID = 3, StandardName= "Standard 3" } }; var innerJoin = studentList.Join( // outer sequence standardList, // inner sequence student => student.StandardID, // outerKeySelector standard => standard.StandardID, // innerKeySelector (student, standard) => new // result selector { StudentName = student.StudentName, StandardName = standard.StandardName });

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-8nUDoV&usg=ALkJrhjn5ZvIv75tBTBPko1MNKfVkm-3yQ)

La siguiente imagen ilustra las partes del operador Join en el ejemplo anterior.

[](http://www.tutorialsteacher.com/Content/images/linq/linq-join-operator.png)operador de unión

En el ejemplo anterior de la consulta de combinación, studentList es una secuencia externa porque la consulta comienza desde allí. El primer parámetro en el método Join se usa para especificar la secuencia interna que es standardList en el ejemplo anterior. El segundo y tercer parámetro del método Join se usa para especificar un campo cuyo valor debe coincidir con la expresión lambda para incluir un elemento en el resultado. El selector de clave para la secuencia externa student => student.StandardID indica que el campo Take ID de cada elemento de studentList debe coincidir con la clave de la secuencia interna standard => standard.StandardID . Standard ID. Si el valor de ambos campos clave coincide, entonces incluya ese elemento en el resultado.

El último parámetro en el método de unión es una expresión para formular el resultado. En el ejemplo anterior, el selector de resultados incluye las propiedades StudentName y StandardName de ambas secuencias.

La clave ID estándar de ambas secuencias (colecciones) debe coincidir, de lo contrario, el elemento no se incluirá en el resultado. Por ejemplo, Ron no está asociado con ningún estándar, por lo que Ron no está incluido en la colección de resultados. innerJoinResult en el ejemplo anterior contendría los siguientes elementos después de la ejecución:

John - Estándar 1  
Moin - Estándar 1  
Bill - Estándar 2  
Ram - Estándar 2

El siguiente ejemplo demuestra el operador Join en la sintaxis del método en VB.Net.

Ejemplo: Unirse al operador VB.Net

Dim innerJoin = studentList.Join(standardList, Function (s) s.StandardID, Function (std) std.StandardID, Function (s, std) New With { .StudentName = s.StudentName, .StandardName = std.StandardName });

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-4hlooi&usg=ALkJrhhkhKjlIZf1tmbtWAVdLYMH6QMZCw)

### Únase a la sintaxis de consulta:

Unirse al operador en la sintaxis de la consulta funciona ligeramente diferente a la sintaxis del método. Requiere secuencia externa, secuencia interna, selector de clave y selector de resultados.La palabra clave 'on' se usa para el selector de teclas donde el lado izquierdo del operador 'igual' es outerKeySelector y el lado derecho de 'iguales' es innerKeySelector.

Sintaxis: unirse a la sintaxis de la consulta

from ... in outerSequence   
 join ... in innerSequence   
 on outerKey equals innerKey   
 select ...

El siguiente ejemplo del operador Join en la sintaxis de la consulta devuelve una colección de elementos de studentList y standardList si su Student.StandardID y Standard.StandardID coinciden.

Ejemplo: Operador de unión en la sintaxis de consulta C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 13, StandardID =1 }, new Student () { StudentID = 2, StudentName = "Moin" , Age = 21, StandardID =1 }, new Student () { StudentID = 3, StudentName = "Bill" , Age = 18, StandardID =2 }, new Student () { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID =2 }, new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; IList < Standard > standardList = new List < Standard >() { new Standard (){ StandardID = 1, StandardName= "Standard 1" }, new Standard (){ StandardID = 2, StandardName= "Standard 2" }, new Standard (){ StandardID = 3, StandardName= "Standard 3" } }; var innerJoin = from s in studentList // outer sequence join st in standardList //inner sequence on s.StandardID equals st.StandardID // key selector select new { // result selector StudentName = s.StudentName, StandardName = st.StandardName };

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 13, StandardID =1 }, new Student () { StudentID = 2, StudentName = "Moin" , Age = 21, StandardID =1 }, new Student () { StudentID = 3, StudentName = "Bill" , Age = 18, StandardID =2 }, new Student () { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID =2 }, new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; IList < Standard > standardList = new List < Standard >() { new Standard (){ StandardID = 1, StandardName= "Standard 1" }, new Standard (){ StandardID = 2, StandardName= "Standard 2" }, new Standard (){ StandardID = 3, StandardName= "Standard 3" } }; var innerJoin = from s in studentList // outer sequence join st in standardList //inner sequence on s.StandardID equals st.StandardID // key selector select new { // result selector StudentName = s.StudentName, StandardName = st.StandardName };

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-4arutq&usg=ALkJrhhtiwACvvk_6jNaqViNM0V0r5GbAw)

Ejemplo: operador de unión en sintaxis de consulta VB.Net

Dim innerJoin = From s In studentList ' outer sequence Join std In standardList ' inner sequence On s.StandardID Equals std.StandardID ' key selector Select \_ ' result selector StudentName = s.StudentName, StandardName = std.StandardName

Salida:

John - Estándar 1  
Moin - Estándar 1  
Bill - Estándar 2  
Ram - Estándar 2

**Nota:Use el operador igual para hacer coincidir el selector de clave en la sintaxis de la consulta.== no es válido.**



### Puntos para recordar :

1. **Join** y **GroupJoin** se unen a los operadores.
2. **Join** es como una combinación interna de SQL. Devuelve una nueva colección que contiene elementos comunes de dos colecciones que coinciden con las claves.
3. **Join** opera en dos secuencias secuencia interna y secuencia externa y produce una secuencia de resultados.
4. **Unir la** sintaxis de la consulta:   
   from... in outerSequence   
   join... in innerSequence   
   on outerKey equals innerKey   
   select ...

Aprenda sobre GroupJoin a continuación.

# Operador que se une: GroupJoin

Hemos visto el operador Join en la sección anterior. El operador GroupJoin realiza la misma tarea que el operador Join, excepto que GroupJoin devuelve un resultado en un grupo según la clave de grupo especificada. El operador GroupJoin une dos secuencias en función de la clave y agrupa el resultado haciendo coincidir la clave y luego devuelve la colección de resultado y clave agrupados.

## GroupJoin in Method Sintaxis:

GroupJoin requiere los mismos parámetros que Join. GroupJoin tiene los siguientes dos métodos de sobrecarga:

Sobrecargas del método Join ():

public static IEnumerable <TResult> GroupJoin <Touter, TInner, TKey, TResult> (esto IEnumerable <Touter> exterior, IEnumerable <TInner> interior, Func <Touter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <Touter, IEnumerable <TInner>, TResult> resultSelector);

public static IEnumerable <TResult> GroupJoin <Touter, TInner, TKey, TResult> (esto IEnumerable <Touter> exterior, IEnumerable <TInner> interior, Func <Touter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <Touter, IEnumerable <TInner>, TResult> resultSelector , IEqualityComparer <TKey> comparer);

Como puede ver en el primer método de sobrecarga, toma cinco parámetros de entrada (excepto el primer parámetro 'this'): 1) externo 2) interno 3) outerKeySelector 4) innerKeySelector 5) resultSelector. Tenga en cuenta que resultSelector es del tipo delegado Func que tiene el segundo parámetro de entrada como tipo IEnumerable para la secuencia interna.

Ahora, comprendamos GroupJoin usando las siguientes clases de Estudiante y Estándar, donde la clase de Estudiante incluye la ID Estándar que coincide con la ID Estándar de la clase Estándar.

Clases de ejemplo

public class Student { public int StudentID { get; set; } public string StudentName { get; set; } public int StandardID { get; set; } } public class Standard { public int StandardID { get; set; } public string StandardName { get; set; } }

Considere el siguiente ejemplo de consulta de GroupJoin.

Ejemplo: GroupJoin in Method sintaxis C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , StandardID =1 }, new Student () { StudentID = 2, StudentName = "Moin" , StandardID =1 }, new Student () { StudentID = 3, StudentName = "Bill" , StandardID =2 }, new Student () { StudentID = 4, StudentName = "Ram" , StandardID =2 }, new Student () { StudentID = 5, StudentName = "Ron" } }; IList < Standard > standardList = new List < Standard >() { new Standard (){ StandardID = 1, StandardName= "Standard 1" }, new Standard (){ StandardID = 2, StandardName= "Standard 2" }, new Standard (){ StandardID = 3, StandardName= "Standard 3" } }; var groupJoin = standardList.GroupJoin(studentList, //inner sequence std => std.StandardID, //outerKeySelector s => s.StandardID, //innerKeySelector (std, studentsGroup) => new // resultSelector { Students = studentsGroup, StandarFulldName = std.StandardName }); foreach ( var item in groupJoin) { Console .WriteLine(item.StandarFulldName ); foreach ( var stud in item.Students) Console .WriteLine(stud.StudentName); }

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-ig99cR&usg=ALkJrhjIShcbh-YuYrk76SlKIyNiW0Jl3g)

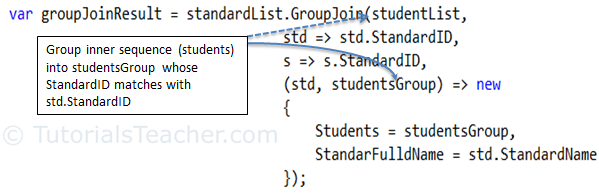
Salida:

Estándar 1:  
John,  
Moin,  
Estándar 2:  
Cuenta,  
RAM,  
Estándar 3:

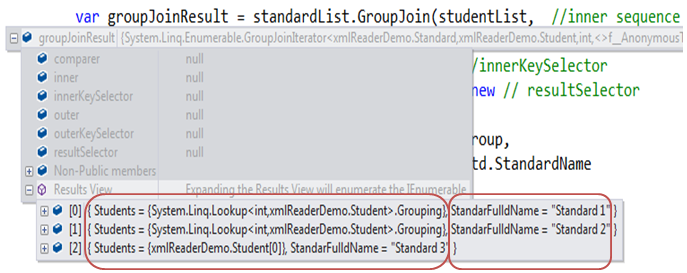
En el ejemplo anterior de la consulta GroupJoin, standardList es la secuencia externa, porque la consulta comienza desde allí. El primer parámetro en el método GroupJoin es especificar la secuencia interna, que es studentList en el ejemplo anterior. El segundo y el tercer parámetro del método GroupJoin () son para especificar un campo cuyo valor debe coincidir utilizando la expresión lambda, para incluir un elemento en el resultado. El selector de clave para el standard => standard.StandardID secuencia externa standard => standard.StandardID indica que el campo StandardID de cada elemento en la lista estándar debe coincidir con la clave de la secuencia interna studentList student => student.StandardID . Si el valor de ambos campos de claves coincide, entonces incluya ese elemento en los grupos de estudiantes agrupados de la colección donde la clave sería StandardID.

El último parámetro en el método de unión es una expresión para formular el resultado. En el ejemplo anterior, el selector de resultados incluye la colección agrupada studentGroup y StandardName.

La siguiente imagen ilustra esa secuencia interna agrupada en StudentsGroup collection para hacer coincidir la clave de StandardID y esa colección agrupada se puede utilizar para formular el resultado.

[](http://www.tutorialsteacher.com/Content/images/linq/linq-groupjoin.png)Operador de Agrupación - GroupJoin

El conjunto de resultados incluiría un objeto anónimo que tiene las propiedades Students y StandardFullName. La propiedad de los estudiantes será una colección de estudiantes cuya StandardID coincida con Standard.StandardID.

[](http://www.tutorialsteacher.com/Content/images/linq/groupJoin-result.png)Resultado de GroupJoin en la vista de depuración

Puede acceder al resultado utilizando un ciclo 'foreach'. Cada elemento tendrá la propiedad StandardFullName & Students, donde Students será una colección.

Ejemplo: acceder a GroupJoin Result en C #

foreach ( var item in groupJoinResult) { Console .WriteLine(item.StandarFulldName ); foreach ( var stud in item.Students) Console .WriteLine(stud.StudentName); }

A continuación se muestra un ejemplo de GroupJoin en VB.Net:

Ejemplo: GroupJoin in Method sintaxis VB.Net

Dim groupJoin = standardList.GroupJoin( ' outer sequence studentList, ' inner sequence Function (s) s.StandardID, ' outerKeySelector Function (stud) stud.StandardID, ' innerKeySelector Function (s, studentGroup) New With { ' result selector .students = studentGroup, .standardName = s.StandardName }) For Each item In groupJoin Console .WriteLine(item.standardName) For Each std In item.students Console .WriteLine( std.StudentName) Next Next

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-WIlZTs&usg=ALkJrhi6TnN76b8xYBVvviShL6k0vwz5_A)

Salida:

Estándar 1:  
John,  
Moin,  
Estándar 2:  
Cuenta,  
RAM,  
Estándar 3:

## GroupJoin in Query Syntax:

El operador GroupJoin en sintaxis de consulta funciona ligeramente diferente de la sintaxis del método. Requiere una secuencia externa, una secuencia interna, un selector de teclas y un selector de resultados. La palabra clave 'on' se usa para el selector de teclas donde el lado izquierdo del operador 'igual' es outerKeySelector y el lado derecho de 'iguales' es el innerKeySelector. Use la palabra clave **into** para crear la colección agrupada.

Sintaxis: GroupJoin in Query sintaxis

from ... in outerSequence   
 join ... in innerSequence   
 on outerKey equals innerKey   
 into groupedCollection   
 select ...

El siguiente ejemplo demuestra el GroupJoin en sintaxis de consulta.

Ejemplo: Sintaxis de consulta de GroupJoin C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 13, StandardID =1 }, new Student () { StudentID = 2, StudentName = "Moin" , Age = 21, StandardID =1 }, new Student () { StudentID = 3, StudentName = "Bill" , Age = 18, StandardID =2 }, new Student () { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID =2 }, new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; IList < Standard > standardList = new List < Standard >() { new Standard (){ StandardID = 1, StandardName= "Standard 1" }, new Standard (){ StandardID = 2, StandardName= "Standard 2" }, new Standard (){ StandardID = 3, StandardName= "Standard 3" } }; var groupJoin = from std in standardList join s in studentList on std.StandardID equals s.StandardID into studentGroup select new { Students = studentGroup , StandardName = std.StandardName }; foreach ( var item in groupJoin) { Console .WriteLine(item.StandarFulldName ); foreach ( var stud in item.Students) Console .WriteLine(stud.StudentName); }

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-307N2j&usg=ALkJrhhPvHz_Ixryy1aIT2wPj2ENpWbrig)

Ejemplo: sintaxis de consulta de GroupJoin VB.Net

Dim groupJoin = From s In standardList Group Join stud In studentList On stud.StandardID Equals s.StandardID Into Group \_ Select \_ StudentsGroup = Group, StandardName = s.StandardName

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-c6sM6D&usg=ALkJrhh8CnfAT7Ty2HYPch9QLVaaEU22xg)

Salida:

Estándar 1:  
John,  
Moin,  
Estándar 2:  
Cuenta,  
RAM,  
Estándar 3:

En VB.Net, la palabra clave **InTo** creará un grupo de todos los estudiantes del mismo estándar y lo asignará a la palabra clave **Group** . Entonces, usa Grupo en el resultado de la proyección.

**Nota:uso del operador igual para emparejar el selector de llave.== no es válido.**

# Operadores de proyección: seleccione, seleccioneMuchos

Hay dos operadores de proyección disponibles en LINQ. 1) Seleccione 2) SeleccioneMany

## Seleccionar:

El operador Seleccionar siempre devuelve una colección IEnumerable que contiene elementos basados ​​en una función de transformación. Es similar a la cláusula Select de SQL que produce un conjunto de resultados plano.

Ahora, comprendamos el operador de consulta Select utilizando la siguiente clase de Estudiante.

Clases de ejemplo

public class Student { public int StudentID { get; set; } public string StudentName { get; set; } public int Age { get; set; } }

### Seleccionar en la sintaxis de la consulta:

La sintaxis de la consulta de LINQ debe finalizar con una cláusula **Select** o **GroupBy** . El siguiente ejemplo muestra el operador de selección que devuelve una colección de cadenas de StudentName.

Ejemplo: Seleccione en la sintaxis de la consulta C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" }, new Student () { StudentID = 2, StudentName = "Moin" }, new Student () { StudentID = 3, StudentName = "Bill" }, new Student () { StudentID = 4, StudentName = "Ram" }, new Student () { StudentID = 5, StudentName = "Ron" } }; var selectResult = from s in studentList select s.StudentName;

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-AS7xgc&usg=ALkJrhjJO8WoDlz2PMRqbhnHFGWoFeSsyw)

El operador de selección se puede utilizar para formular el resultado según nuestro requisito. Se puede usar para devolver una colección de clase personalizada o tipo anónimo que incluye propiedades según nuestras necesidades.

El siguiente ejemplo de la cláusula select devuelve una colección de [tipo anónimo que](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/csharp/csharp-anonymous-type&usg=ALkJrhgmW_jQjR_jLcdcsS5Y8HSsOih0AA) contiene la propiedad Nombre y Edad.

Ejemplo: Seleccionar operador en sintaxis de consulta C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 13 } , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; // returns collection of anonymous objects with Name and Age property var selectResult = from s in studentList select new { Name = "Mr. " + s.StudentName, Age = s.Age }; // iterate selectResult foreach ( var item in selectResult) Console .WriteLine( "Student Name: {0}, Age: {1}" , item.Name, item.Age);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-Eq53Zo&usg=ALkJrhgaA803S4HmIZIjHteD0GZYp6AlfQ)

Ejemplo: Seleccionar operador en sintaxis de consulta VB.Net

Dim selectResult = From s In studentList Select New With {.Name = s.StudentName, .Age = s.Age}

Salida:

Nombre del estudiante: Sr. John, Edad: 13  
Nombre del estudiante: Sr. Moin, Edad: 21  
Nombre del estudiante: Sr. Bill, Edad: 18  
Nombre del estudiante: Sr. Ram, Edad: 20  
Nombre del estudiante: Sr. Ron, Edad: 15

### Seleccione en Método de sintaxis:

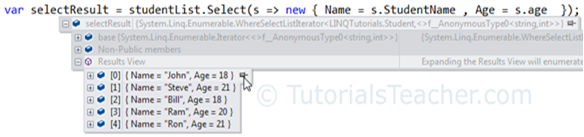
El operador Seleccionar es opcional en la sintaxis del método. Sin embargo, puede usarlo para dar forma a los datos. En el siguiente ejemplo, el método de extensión Select devuelve una colección de objetos anónimos con la propiedad Nombre y Edad:

Ejemplo: Seleccione en la sintaxis del método C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 21 } }; var selectResult = studentList.Select(s => new { Name = s.StudentName , Age = s.Age });

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-apzWfe&usg=ALkJrhjOTuvlJtRW8zUJReSfGMpk1lfhtg)

En el ejemplo anterior, selectResult contendría objetos anónimos con propiedades Name y Age como se muestra a continuación en la vista de depuración.

[](http://www.tutorialsteacher.com/Content/images/linq/lnq-select-result.png)La cláusula Select devuelve un objeto anónimo

Ejemplo: Seleccione en la sintaxis del método VB.Net

Dim selectResult = studentList. Select ( Function (s) New With {.Name = s.StudentName, .Age = s.Age})

## Seleccione muchos:

El operador SelectMany proyecta secuencias de valores que se basan en una función de transformación y luego las aplana en una secuencia.

Visite MSDN para obtener más información sobre los [operadores de proyección](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://msdn.microsoft.com/en-us/library/bb546168.aspx%3Fcs-save-lang%3D1%26cs-lang%3Dcsharp&usg=ALkJrhg1Dl1W6zu4rnkMlfwHiqm0KQyR5g#code-snippet-1) .

# Operadores de cuantificadores:

Los operadores del cuantificador evalúan los elementos de la secuencia en alguna condición y devuelven un valor booleano para indicar que algunos o todos los elementos satisfacen la condición.

| **Operador** | **Descripción** |
| --- | --- |
| Todas | Comprueba si todos los elementos en una secuencia satisfacen la condición especificada |
| Alguna | Comprueba si alguno de los elementos en una secuencia satisface la condición especificada |
| Contiene | Comprueba si la secuencia contiene un elemento específico |

## Todas:

El operador Todo evalúa cada elemento de la colección dada en una condición específica y devuelve Verdadero si todos los elementos satisfacen una condición.

Ejemplo: Todo el operador C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; // checks whether all the students are teenagers bool areAllStudentsTeenAger = studentList.All(s => s.Age > 12 && s.Age < 20); Console.WriteLine(areAllStudentsTeenAger);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-sFOdZM&usg=ALkJrhjvENRKxr_l-UXBz8s-o9-dkzliig)

Ejemplo: todos los operadores VB.Net

Dim areAllStudentsTeenAger = studentList.All( Function (s) s.Age > 12 And s.Age < 20)

Salida:

falso

## Alguna:

¿Cualquier control comprueba si algún elemento cumple con la condición dada o no? En el siguiente ejemplo, cualquier operación se usa para verificar si un alumno es adolescente o no.

Ejemplo: cualquier operador C #

bool isAnyStudentTeenAger = studentList.Any(s => s.age > 12 && s.age < 20);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-NDBYRk&usg=ALkJrhhZfYocyJfF3D3HwI6MCG_MNthdyw)

Ejemplo: cualquier operador VB.Net

Dim isAnyStudentTeenAger = studentList.Any( Function (s) s.Age > 12 And s.Age < 20)

Salida:

cierto

**Nota: Losoperadores de cuantificadores no son compatibles con la sintaxis de consulta de C #.**

Obtenga información sobre el operador de cuantificador: contiene la siguiente sección.

# Operador de cuantificador: contiene

El operador Contiene comprueba si un elemento especificado existe en la colección o no y devuelve un booleano.

El método de extensión Contains () tiene dos sobrecargas siguientes. El primer método de sobrecarga requiere un valor para verificar en la colección y el segundo método de sobrecarga requiere un parámetro adicional del tipo IEqualityComparer para la comparación de equalalidad personalizada.

Sobrecargas del método Contiene ():

public static bool Contiene <TSource> ( esta fuente IEnumerable <TSource>, valor de TSource);

public static bool Contiene <TSource> ( esta fuente IEnumerable <TSource>,

Valor de Source,

IEqualityComparer <TSource> comparador);

Como se mencionó anteriormente, el método de extensión Contains () requiere un valor para verificar como un parámetro de entrada. El tipo de valor debe ser el mismo que el tipo de colección genérica. El siguiente ejemplo de contiene comprueba si 10 existe en la colección o no. Tenga en cuenta que int es un tipo de colección genérica.

Ejemplo: Contiene el operador C #

IList < int > intList = new List < int >() { 1, 2, 3, 4, 5 }; bool result = intList.Contains(10); // returns false

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-7QLxOl&usg=ALkJrhhnYUFDAfUwtGYtaFwIqmyTBoltTA)

Ejemplo: Contiene el operador VB.Net

Dim intList As IList(Of Integer ) = New List(Of Integer ) From {1, 2, 3, 4, 5} Dim result = intList.Contains(10) ' returns false

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-ef5kJq&usg=ALkJrhgJ2asD1hP8dTM8YYJS37ORF_Ae8w)

El ejemplo anterior funciona bien con tipos de datos primitivos. Sin embargo, no funcionará con una clase personalizada. Considere el siguiente ejemplo:

DO#:

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; Student std = new Student (){ StudentID =3, StudentName = "Bill" }; bool result = studentList.Contains(std); //returns false

Como puede ver en el ejemplo anterior, Contains devuelve false incluso si "Bill" existe en la lista de estudiantes. Esto se debe a que el método de extensión Contiene solo compara la referencia de un objeto pero no los valores reales de un objeto. Entonces, para comparar valores del objeto estudiante, necesita crear una clase implementando la interfaz IEqualityComparer, que compara valores de dos objetos Student y devuelve booleano.

La siguiente es una clase StudentComparer que implementa la interfaz IEqualityComparer <Student> para comparar valores de dos objetos de Students:

Ejemplo: IEqualityComperer

class StudentComparer : IEqualityComparer < Student > { public bool Equals( Student x, Student y) { if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower()) return true ; return false ; } public int GetHashCode( Student obj) { return obj.GetHashCode(); } }

Ahora, puede usar la clase StudentComparer anterior en el segundo método de sobrecarga de método de extensión Contains que acepta el segundo parámetro del tipo IEqualityComparer, como se muestra a continuación:

Ejemplo: contiene con Comparer clase C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; Student std = new Student (){ StudentID =3, StudentName = "Bill" }; bool result = studentList.Contains(std, new StudentComparer ()); // **returns true**

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-EVTaLz&usg=ALkJrhhB_pUEMeyxKteCCou9zGZJyiDEIQ)

Por lo tanto, debe usar la clase comparer para obtener el resultado corrent del método de extensión Contains para clases personalizadas.

El siguiente es un ejemplo similar en VB.Net:

Ejemplo: contiene una clase Comparer VB.Net

public class Student { public int StudentID { get ; set ; } public string StudentName { get ; set ; } public int Age { get ; set ; } } Public Class StudentComparer Implements IEqualityComparer(Of Student) Public Function Equals1(x As Student, y As Student) As Boolean Implements IEqualityComparer(Of Student).Equals If (x.StudentID = y.StudentID And x.StudentName.ToLower() = y.StudentName.ToLower()) Then Return True End If Return False End Function Public Function GetHashCode1(obj As Student) As Integer Implements IEqualityComparer(Of Student).GetHashCode Return obj.GetHashCode() End Function End Class Sub Main Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 18}, New Student () With {.StudentID = 2, .StudentName = "Steve" , .Age = 15}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 25}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 19} } Dim std As New Student With {.StudentID = 3, .StudentName = "Bill" } Dim result = studentList.Contains(std, New StudentComparer()) ' returns true End Sub

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-lmPMdE&usg=ALkJrhgQ_lql9iP6YlaYjNZnRF_ft4kOVw)

**Nota: Losoperadores de cuantificadores no son compatibles con la sintaxis de consulta en C # o VB.Net.**



### Puntos para recordar:

1. All, Any & Contains son operadores cuantificadores en LINQ.
2. Todos los controles si todos los elementos en una secuencia satisfacen la condición especificada.
3. Cualquier comprobación si alguno de los elementos en una secuencia satisface la condición especificada
4. Contiene operadores que comprueban si el elemento especificado existe en la colección o no.
5. Use una clase personalizada que derive IEqualityOperator con Contiene para verificar el objeto en la colección.
6. All, Any y Contains no son compatibles con la sintaxis de consulta en C # o VB.Net.

Más información sobre el operador agregado.

# Operadores de Agregación: Agregado

Los operadores de agregación realizan operaciones matemáticas como promedio, agregado, recuento, máximo, mínimo y suma, en la propiedad numérica de los elementos en la colección.

| **Método** | **Descripción** |
| --- | --- |
| Agregar | Realiza una operación de agregación personalizada en los valores de la colección. |
| Promedio | calcula el promedio de los elementos numéricos en la colección. |
| Contar | Cuenta los elementos en una colección. |
| LongCount | Cuenta los elementos en una colección. |
| Max | Encuentra el valor más grande en la colección. |
| Min. | Encuentra el valor más pequeño en la colección. |
| Suma | Calcula la suma de los valores en la colección. |

## Agregar:

El método Agregado realiza una operación de acumulación. El método de extensión agregada tiene los siguientes métodos de sobrecarga:

Sobrecargas del método Aggregate ():

public static TSource Aggregate <TSource> (esta fuente IEnumerable <TSource>,

Func <TSource, TSource, TSource> func);

public static TAccumulate Aggregate <TSource, TAccumulate> (esta fuente IEnumerable <TSource>,

TAcumular semilla,

Func <TAccumulate, TSource, TAcumulate> func);

public static TResult Aggregate <TSource, TAccumulate, TResult> (esta fuente IEnumerable <TSource>,

TAcumular semilla,

Func <TAccumulate, TSource, TAccumulate> func,

Func <TAccumulate, TResult> resultSelector);

El siguiente ejemplo muestra el método Agregado que devuelve elementos separados por comas de la lista de cadenas.

Ejemplo: operador agregado en la sintaxis del método C #

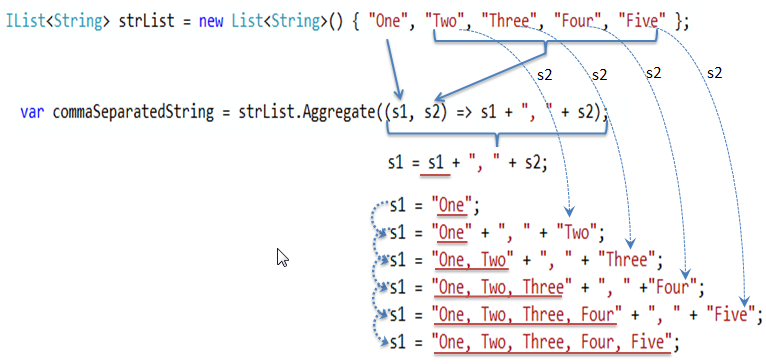
IList < String > strList = new List < String >() { "One" , "Two" , "Three" , "Four" , "Five" }; var commaSeperatedString = strList.Aggregate((s1, s2) => s1 + ", " + s2); Console .WriteLine(commaSeperatedString);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-zJ8Lxb&usg=ALkJrhjmNFQ0CHTi6QzWBZ9lmpyMWkipZA)

Salida:

Uno dos tres CUATRO CINCO

En el ejemplo anterior, el método de extensión agregada devuelve cadenas separadas por comas de la colección strList. La siguiente imagen ilustra toda la operación agregada realizada en el ejemplo anterior.

[](http://www.tutorialsteacher.com/Content/images/linq/linq-aggregate-1.png)Método de extensión agregada

Según la figura anterior, el primer ítem de strList "One" pasará como s1 y el resto de los ítems se pasará como s2. La expresión lambda (s1, s2) => s1 + ", " + s2 se tratará como s1 = s1 + ", " + s1 donde s1 se acumulará para cada elemento de la colección. Por lo tanto, el método Agregado devolverá una cadena separada por comas.

Ejemplo: operador agregado en la sintaxis del método VB.Net

Dim strList As IList(Of String ) = New List(Of String ) From { "One" , "Two" , "Three" , "Four" , "Five" } Dim commaSeparatedString = strList.Aggregate( Function (s1, s2) s1 + ", " + s2)

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-hh6qto&usg=ALkJrhg966wEGahWD_zFGJ0y2HKmfJEB3w)

## Método agregado con valor de inicialización:

El segundo método de sobrecarga de Aggregate requiere que se acumule el primer parámetro para el valor inicial. El segundo parámetro es el delegado de tipo Func:   
TAccumulate Aggregate<TSource, TAccumulate>( **TAccumulate seed, Func<TAccumulate, TSource, TAccumulate> func** ); .

El siguiente ejemplo usa una cadena como valor inicial en el método de extensión Agregado.

Ejemplo: método agregado con valor de inicialización en C #

// Student collection IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; string commaSeparatedStudentNames = studentList.Aggregate< Student , string >( "Student Names: " , // seed value (str, s) => str += s.StudentName + "," ); Console .WriteLine(commaSeparatedStudentNames);

// Student collection IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; string commaSeparatedStudentNames = studentList.Aggregate< Student , string >( "Student Names: " , // seed value (str, s) => str += s.StudentName + "," ); Console .WriteLine(commaSeparatedStudentNames);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-vqMaxP&usg=ALkJrhgjPqsmJmsGz4swOtCibicP2Wt_Rg)

Ejemplo: método agregado con valor inicial en VB.Net

// Student collection Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 13}, New Student () With {.StudentID = 2, .StudentName = "Moin" , .Age = 21}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 18}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 15} } Dim commaSeparatedStudentNames = studentList.Aggregate(Of String )( "Student Names: " , Function (str, s) str + s.StudentName + "," ) Console .WriteLine(commaSeparatedStudentNames);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-UuxdRS&usg=ALkJrhhkj-vn58ZLwkfBjNEfCME2N0ZGHA)

Salida:

Nombres de los estudiantes: John, Moin, Bill, Ram, Ron,

En el ejemplo anterior, el primer parámetro del método Aggregate es la cadena "Nombres de estudiantes:" que se acumulará con todos los nombres de los alumnos. La coma en la expresión lambda se pasará como un segundo parámetro.

El siguiente ejemplo usa el operador Agregado para agregar la edad de todos los estudiantes.

Ejemplo: método agregado con valor de inicialización en C #

// Student collection IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; int SumOfStudentsAge = studentList.Aggregate< Student , int >(0, (totalAge, s) => totalAge += s.Age );

// Student collection IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; int SumOfStudentsAge = studentList.Aggregate< Student , int >(0, (totalAge, s) => totalAge += s.Age );

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-l0Q8bw&usg=ALkJrhgiqw9IVz0jlMt29AEObqkNFmZ6Ug)

## Método agregado con selector de resultados:

Ahora, veamos el tercer método de sobrecarga que requiere el tercer parámetro de la expresión del delegado de Func para el selector de resultados, para que puedas formular el resultado.

Considera el siguiente ejemplo.

Ejemplo: método agregado C #

IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; string commaSeparatedStudentNames = studentList.Aggregate< Student , string , string >( String .Empty, // seed value (str, s) => str += s.StudentName + "," , // returns result using seed value, String.Empty goes to lambda expression as str str => str.Substring(0,str.Length - 1 )); // result selector that removes last comma Console .WriteLine(commaSeparatedStudentNames);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-eEFeYx&usg=ALkJrhj38pUfHCqKrZdSXGvxLiHpsb571Q)

En el ejemplo anterior, hemos especificado una expresión lambda str => str.Substring(0,str.Length - 1 ) que eliminará la última coma en el resultado de la cadena. A continuación se muestra el mismo ejemplo en VB.Net.

Ejemplo: método agregado VB.Net

// Student collection Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 13}, New Student () With {.StudentID = 2, .StudentName = "Moin" , .Age = 21}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 18}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 15} } Dim commaSeparatedStudentNames = studentList.Aggregate(Of String , String )( String .Empty, Function (str, s) str + s.StudentName + "," , Function (str) str.Substring(0, str.Length - 1)) Console .WriteLine(commaSeparatedStudentNames);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-44HMz5&usg=ALkJrhjBWBQEiGoM-BMXzXj1_wWdQFxW8A)

Salida:

John, Moin, Bill, Ram, Ron

**Nota:El operador agregado no es compatible con la sintaxis de consulta en C # o VB.Net.**

Obtenga información sobre otro operador agregado: promedio en la siguiente sección.

# Operador de Agregación: Promedio

El método de extensión promedio calcula el promedio de los elementos numéricos en la colección.El método promedio arroja valores decimales, dobles o flotantes que aceptan nulos o que no aceptan nulos.

El siguiente ejemplo muestra el método de Agerage que devuelve el valor promedio de todos los enteros de la colección.

Ejemplo: método promedio - C #

IList < int > intList = new List< int >>() { 10, 20, 30 }; var avg = intList.Average(); Console .WriteLine( "Average: {0}" , avg);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-fPDXJu&usg=ALkJrhhH_1U64CDcDtgkWv58t0b_1StogQ)

Puede especificar una propiedad int, decimal, doble o flotante de una clase como expresión lambda de la cual desea obtener un valor promedio. El siguiente ejemplo muestra el método Promedio en el tipo complejo.

Ejemplo: operador promedio en la sintaxis del método C #

IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; var avgAge = studentList.Average(s => s.Age); Console .WriteLine( "Average Age of Student: {0}" , avgAge);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-y4elbo&usg=ALkJrhhY0bTlj4VWE2mcPMCRKYJaPbdeMw)

Ejemplo: operador promedio en sintaxis de método VB.Net

Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 13}, New Student () With {.StudentID = 2, .StudentName = "Moin" , .Age = 21}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 18}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 15} } Dim avgAge = studentList.Average( Function (s) s.Age) Console .WriteLine( "Average Age of Student: {0}" , avgAge)

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-NBjXY9&usg=ALkJrhhXV9IJePw6DkMbvcpcoNLQQWdk1Q)

Salida:

Promedio de edad del estudiante: 17,4

El operador promedio en la sintaxis de consulta **no** es **compatible** con C #. Sin embargo, es compatible con VB.Net como se muestra a continuación.

Ejemplo: operador promedio en sintaxis de consulta VB.Net

Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 13}, New Student () With {.StudentID = 2, .StudentName = "Moin" , .Age = 21}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 18}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 15} } Dim avgAge = Aggregate st In studentList Into Average(st.Age) Console .WriteLine( "Average Age of Student: {0}" , avgAge)

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-J6WaJN&usg=ALkJrhjruR5Do8ra3QsWg_VkGBZcdAHBlg)

Salida:

Promedio de edad del estudiante: 17,4

Obtenga información sobre otro operador agregado: cuente en la siguiente sección.

# Operador de Agregación: Count

El operador de recuento devuelve la cantidad de elementos en la colección o la cantidad de elementos que han satisfecho la condición dada.

El método de extensión Count () tiene las siguientes dos sobrecargas:

Las sobrecargas del método Count () en C #

int Count<TSource>(); int Count<TSource>(Func<TSource, bool > predicate);

El primer método de sobrecarga de Count devuelve el número de elementos en la colección especificada, mientras que el segundo método de sobrecarga devuelve la cantidad de elementos que han satisfecho la condición especificada dada como expresión lambda / función de predicado.

El siguiente ejemplo devuelve la cantidad de elementos en una colección utilizando el método Count () en la sintaxis del método.

Ejemplo: Count () en C #

IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Mathew" , Age = 15 } }; var numOfStudents = studentList.Count(); Console .WriteLine( "Number of Students: {0}" , numOfStudents);

Ejemplo: Count () en VB.Net

Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 13}, New Student () With {.StudentID = 2, .StudentName = "Moin" , .Age = 21}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 18}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 15} } Dim numOfStudents = studentList.Count() Console .WriteLine( "Number of Students: {0}" , numOfStudents);

Salida:

Cantidad de estudiantes: 5

En el siguiente ejemplo, obtenemos el recuento de estudiantes cuya edad es 18 o más al especificar la condición en el método Contar:

DO#:

// Student collection IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Mathew" , Age = 15 } }; var numOfStudents = studentList.Count(s => s.Age >= 18); Console .WriteLine( "Number of Students: {0}" , numOfStudents);

Salida:

Cantidad de estudiantes: 3

**Nota: Elmétodo de extensión Count () con el parámetro de predicado no es compatible en VB.Net.**

## Operador de recuento en sintaxis de consulta:

Ejemplo: Operador de recuento en sintaxis de consulta VB.Net

// Student collection Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 13}, New Student () With {.StudentID = 2, .StudentName = "Moin" , .Age = 21}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 18}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 15} } Dim numOfStudents = Aggregate st In studentList Into Count(st.Age >= 18) Console .WriteLine( "Number of Student: {0}" , numOfStudents);

Salida:

Cantidad de estudiantes: 3

La sintaxis de consulta de C # no admite operadores de agregación. Sin embargo, puede ajustar la consulta entre corchetes y usar una función de agregación como se muestra a continuación.

Ejemplo: Operador de recuento en sintaxis de consulta C #

var totalAge = ( from s in studentList select s.age).Count();

Obtenga información sobre otro operador agregado: Max en la siguiente sección.

# Operador de Agregación: Max

El operador Max devuelve el elemento numérico más grande de una colección.

El siguiente ejemplo muestra Max () en la colección primitiva.

Ejemplo: método máximo - C #

IList < int > intList = new List< int >() { 10, 21, 30, 45, 50, 87 }; var largest = intList.Max(); Console .WriteLine( "Largest Element: {0}" , largest); var largestEvenElements = intList.Max(i => { if (i%2 == 0) return i; return 0; }); Console .WriteLine( "Largest Even Element: {0}" , largestEvenElements );

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-4zy5a7&usg=ALkJrhgACJQR-rU6_doZMU5eSKDrcE6Lig)

Salida:

Elemento más grande: 87  
Elemento par más grande: 50

El siguiente ejemplo muestra el método Max () en la colección de tipos complejos.

Ejemplo: operador Max en la sintaxis del método C #

IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; var oldest = studentList.Max(s => s.Age); Console .WriteLine( "Oldest Student Age: {0}" , oldest);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-j73zRR&usg=ALkJrhhFPsUx7RoE0rq8B6EsePHZrtweHw)

Ejemplo: operador Max en la sintaxis del método VB.Net

Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 13}, New Student () With {.StudentID = 2, .StudentName = "Moin" , .Age = 21}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 18}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 15} } Dim oldest = studentList.Max( Function (s) s.Age) Console .WriteLine( "Oldest Student Age: {0}" , oldest)

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-M2qcI3&usg=ALkJrhi-1h17gWKWtASA9xUW_gf7fl5fiQ)

Salida:

Estudiante más viejo Ag: 21

Max devuelve un resultado de cualquier tipo de datos. El siguiente ejemplo muestra cómo puedes encontrar un estudiante con el nombre más largo en la colección:

C #: LINQ Max ()

public class Student : IComparable < Student > { public int StudentID { get ; set ; } public string StudentName { get ; set ; } public int Age { get ; set ; } public int StandardID { get ; set ; } public int CompareTo( Student other) { if ( this .StudentName.Length >= other.StudentName.Length) return 1; return 0; } } class Program { static void Main( string [] args) { // Student collection IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Steve" , Age = 15 } }; var studentWithLongName = studentList.Max(); Console .WriteLine( "Student ID: {0}, Student Name: {1}" , .StudentID, studentWithLongName.StudentName) } }

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-pzp6uz&usg=ALkJrhgT-WGwkxnbhC6801oTvoPOoXP0cA)

Salida:

ID del estudiante: 5, Nombre del estudiante: Steve

**Nota:Puede usar el método / operador de extensión Min del mismo modo que Max.**

Según el ejemplo anterior, para encontrar al estudiante con el nombre más largo, debe implementar la interfaz <T> de IComparable y comparar la longitud de los nombres de los estudiantes en el método CompareTo. Entonces, ahora puede usar el método Max () que usará el método CompareTo para devolver el resultado apropiado.

## Operador Max en la sintaxis de la consulta:

El operador Max **no** es **compatible** con la sintaxis de Consulta C #. Sin embargo, es compatible con la sintaxis de consulta VB.Net como se muestra a continuación.

Ejemplo: operador Max en sintaxis de consulta VB.Net

Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 13}, New Student () With {.StudentID = 2, .StudentName = "Moin" , .Age = 21}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 18}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 15} } Dim maxAge = Aggregate st In studentList Into Max(st.Age) Console .WriteLine( "Maximum Age of Student: {0}" , maxAge);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-msuMmw&usg=ALkJrhiiw3jkSTTtahTNnJSxmEBlCzUTNw)

Salida:

Edad máxima del estudiante: 21

Obtenga información sobre otro operador agregado: Suma en la siguiente sección.

# Operador de Agregación: Suma

El método Sum () calcula la suma de elementos numéricos en la colección.

El siguiente ejemplo muestra Sum () en la colección primitiva.

Ejemplo: LINQ Sum () - C #

IList < int > intList = new List< int >() { 10, 21, 30, 45, 50, 87 }; var total = intList.Sum(); Console .WriteLine( "Sum: {0}" , total); var sumOfEvenElements = intList.Sum(i => { if (i%2 == 0) return i; return 0; }); Console .WriteLine( "Sum of Even Elements: {0}" , sumOfEvenElements );

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-pLrXMc&usg=ALkJrhjAPACtSp935urzoMbBdsAlX4BN5g)

Salida:

Suma: 243  
Suma de elementos pares: 90

El siguiente ejemplo calcula la suma de la edad de todos los estudiantes y también la cantidad de estudiantes adultos en una colección de estudiantes.

Ejemplo: LINQ Sum () - C #

IList < Student > studentList = new List< Student >>() { new Student () { StudentID = 1, StudentName = "John" , Age = 13} , new Student () { StudentID = 2, StudentName = "Moin" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20} , new Student () { StudentID = 5, StudentName = "Ron" , Age = 15 } }; var sumOfAge = studentList.Sum(s => s.Age); Console .WriteLine( "Sum of all student's age: {0}" , sumOfAge); var numOfAdults = studentList.Sum(s => { if (s.Age >= 18) return 1; else return 0; }); Console .WriteLine( "Total Adult Students: {0}" , numOfAdults);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-946u1z&usg=ALkJrhiI14B7jv33hLGY6AmnZ5LBnHmw_w)

Ejemplo: suma en la sintaxis del método - VB.NET

// Student collection Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 13}, New Student () With {.StudentID = 2, .StudentName = "Moin" , .Age = 21}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 18}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 15} } Dim sumOfAge = studentList.Sum( Function (s) s.Age) Console .WriteLine( "Total Age of Student: {0}" , sumOfAge) Dim numOfAdults = studentList.Sum( Function (s) if (s.Age >= 18) return 1 else return 0 end if end function ) Console .WriteLine( "Total Adult Students: {0}" , numOfAdults)

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-JnP4bo&usg=ALkJrhgMiTX39d9auw-721oAjXZb7MXOTg)

Salida:

Edad total del estudiante: 87  
Total de estudiantes adultos: 3

## Operador de suma en sintaxis de consulta:

El operador de suma **no** es **compatible** con la sintaxis de C # Query.

Ejemplo: Operador suma en sintaxis de consulta - VB.Net

// Student collection Dim studentList = New List (Of Student ) From { New Student () With {.StudentID = 1, .StudentName = "John" , .Age = 13}, New Student () With {.StudentID = 2, .StudentName = "Moin" , .Age = 21}, New Student () With {.StudentID = 3, .StudentName = "Bill" , .Age = 18}, New Student () With {.StudentID = 4, .StudentName = "Ram" , .Age = 20}, New Student () With {.StudentID = 5, .StudentName = "Ron" , .Age = 15} } Dim totalAge = Aggregate st In studentList Into Sum(st.Age) Console .WriteLine( "Sum of all student's age: {0}" , totalAge);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-5mY6yW&usg=ALkJrhhx4UK9EXzzxVBrZXIXyBQg4h97Sg)

Salida:

Suma de la edad de todos los estudiantes: 87

# Operadores de elementos: ElementAt, ElementAtOrDefault

Los operadores de elementos devuelven un elemento particular de una secuencia (colección).

La siguiente tabla enumera todos los operadores Elemento en LINQ.

| **Operadores de elementos (métodos)** | **Descripción** |
| --- | --- |
| ElementAt | Devuelve el elemento en un índice especificado en una colección |
| ElementAtOrDefault | Devuelve el elemento en un índice especificado en una colección o un valor predeterminado si el índice está fuera de rango. |
| primero | Devuelve el primer elemento de una colección, o el primer elemento que satisface una condición. |
| FirstOrDefault | Devuelve el primer elemento de una colección, o el primer elemento que satisface una condición. Devuelve un valor predeterminado si el índice está fuera de rango. |
| Último | Devuelve el último elemento de una colección, o el último elemento que satisface una condición |
| LastOrDefault | Devuelve el último elemento de una colección, o el último elemento que satisface una condición. Devuelve un valor predeterminado si no existe dicho elemento. |
| Soltero | Devuelve el único elemento de una colección, o el único elemento que satisface una condición. |
| SingleOrDefault | Devuelve el único elemento de una colección, o el único elemento que satisface una condición. Devuelve un valor predeterminado si no existe dicho elemento o la colección no contiene exactamente un elemento. |

El método ElementAt () devuelve un elemento del índice especificado de una colección determinada. Si el índice especificado está fuera del rango de una colección, arrojará una excepción *Índice fuera de rango* . Tenga en cuenta que el índice es un índice basado en cero.

El método ElementAtOrDefault () también devuelve un elemento del índice especificado de una unión y si el índice especificado está fuera del alcance de una colección, devolverá un valor predeterminado del tipo de datos en lugar de arrojar un error.

El siguiente ejemplo muestra el método ElementAt y ElementAtOrDefault en la colección primitiva.

Ejemplo: LINQ ElementAt () y ElementAtOrDefault () - C #

IList < int > intList = new List< int >() { 10, 21, 30, 45, 50, 87 }; IList < string > strList = new List< string >() { "One" , "Two" , null , "Four" , "Five" }; Console .WriteLine( "1st Element in intList: {0}" , intList.ElementAt(0)); Console .WriteLine( "1st Element in strList: {0}" , strList.ElementAt(0)); Console .WriteLine( "2nd Element in intList: {0}" , intList.ElementAt(1)); Console .WriteLine( "2nd Element in strList: {0}" , strList.ElementAt(1)); Console .WriteLine( "3rd Element in intList: {0}" , intList.ElementAtOrDefault(2)); Console .WriteLine( "3rd Element in strList: {0}" , strList.ElementAtOrDefault(2)); Console .WriteLine( "10th Element in intList: {0} - default int value" , intList.ElementAtOrDefault(9)); Console .WriteLine( "10th Element in strList: {0} - default string value (null)" , strList.ElementAtOrDefault(9)); Console .WriteLine( "intList.ElementAt(9) throws an exception: Index out of range" ); Console .WriteLine( "-------------------------------------------------------------" ); Console .WriteLine(intList.ElementAt(9));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-1Yx1S1&usg=ALkJrhhv4omnRwskQ9P-ji7ojky3EB_CDw)

Salida:

Primer elemento en intList: 10  
Primer elemento en strList: uno  
Segundo elemento en la lista int: 21  
2 ° elemento en strList:  
3er Elemento en intList: 30  
3er Elemento en strList: Tres  
10 ° elemento en intList: 0 - valor int por defecto  
10 ° elemento en strList: - valor de cadena predeterminado (nulo)  
intList.ElementAt (9) arroja una excepción: índice fuera de rango  
-------------------------------------------------- -----------  
Excepción en tiempo de ejecución: el índice estaba fuera de rango ...

Como puede ver en el ejemplo anterior, intList.ElementAtOrDefault(9) devuelve 0 (valor predeterminado de int) porque intList no incluye el décimo elemento. Sin embargo, intList.ElementAt(9) arroja la excepción "Índice fuera de rango". Del mismo modo, strList.ElementAtOrDefault(9) devuelve nulo, que es el valor predeterminado del tipo de cadena. (la consola muestra espacio vacío porque no puede mostrar nulo)

Por lo tanto, es aconsejable utilizar el método de extensión ElementAtOrDefault para eliminar la posibilidad de una excepción de tiempo de ejecución.

Aprenda sobre el operador de otro elemento First y FirstOrDefault en la siguiente sección.

# Operadores de elementos: First & FirstOrDefault

El método First y FirstOrDefault devuelve un elemento del índice zeroth en la colección, es decir, el primer elemento. Además, devuelve un elemento que satisface la condición especificada.

| **Operadores de elementos** | **Descripción** |
| --- | --- |
| primero | Devuelve el primer elemento de una colección, o el primer elemento que satisface una condición. |
| FirstOrDefault | Devuelve el primer elemento de una colección, o el primer elemento que satisface una condición.Devuelve un valor predeterminado si el índice está fuera de rango. |

First y FirstOrDefault tiene dos métodos de sobrecarga. El primer método de sobrecarga no toma ningún parámetro de entrada y devuelve el primer elemento en la colección. El segundo método de sobrecarga toma la expresión lambda como delegado de predicado para especificar una condición y devuelve el primer elemento que satisface la condición especificada.

Métodos de sobrecarga de First y FirstOrDefault - C #

public static TSource First <TSource> ( esta fuente IEnumerable <TSource>);

public static TSource First <TSource> ( esta fuente IEnumerable <TSource>, Func <TSource, bool > predicado);

public static TSource FirstOrDefault <TSource> ( esta fuente IEnumerable <TSource>);

public static TSource FirstOrDefault <TSource> ( esta fuente IEnumerable <TSource>, Func <TSource, bool > predicado);

El método First () devuelve el primer elemento de una colección, o el primer elemento que satisface la condición especificada utilizando la expresión lambda o el delegado Func. Si una colección determinada está vacía o no incluye ningún elemento que satisfaga la condición, arrojará la excepción InvalidOperation.

El método FirstOrDefault () hace lo mismo que el método First (). La única diferencia es que devuelve el valor predeterminado del tipo de datos de una colección si una colección está vacía o no encuentra ningún elemento que satisfaga la condición.

El siguiente ejemplo muestra el método First ().

Ejemplo: LINQ First () - C #

IList < int > intList = new List< int >() { 7, 10, 21, 30, 45, 50, 87 }; IList < string > strList = new List< string >() { null , "Two" , "Three" , "Four" , "Five" }; IList < string > emptyList = new List< string >(); Console .WriteLine( "1st Element in intList: {0}" , intList.First()); Console .WriteLine( "1st Even Element in intList: {0}" , intList.First(i => i % 2 == 0)); Console .WriteLine( "1st Element in strList: {0}" , strList.First()); Console .WriteLine( "emptyList.First() throws an InvalidOperationException" ); Console .WriteLine( "-------------------------------------------------------------" ); Console .WriteLine(emptyList.First());

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-sifJtn&usg=ALkJrhiHY5ID-dBLfCgmvkcb7Lqoh-r1fA)

Salida:

Primer elemento en intList: 7  
1st Even Element en intList: 10  
Primer elemento en strList:  
emptyList.First () arroja una InvalidOperationException  
-------------------------------------------------- -----------  
Excepción de tiempo de ejecución: la secuencia no contiene elementos ...

El siguiente ejemplo muestra el método FirstOrDefault ().

Ejemplo: LINQ FirstOrDefault () - C #

IList < int > intList = new List< int >() { 7, 10, 21, 30, 45, 50, 87 }; IList < string > strList = new List< string >() { null , "Two" , "Three" , "Four" , "Five" }; IList < string > emptyList = new List< string >(); Console .WriteLine( "1st Element in intList: {0}" , intList.FirstOrDefault()); Console .WriteLine( "1st Even Element in intList: {0}" , intList.FirstOrDefault(i => i % 2 == 0)); Console .WriteLine( "1st Element in strList: {0}" , strList.FirstOrDefault()); Console .WriteLine( "1st Element in emptyList: {0}" , emptyList.FirstOrDefault());

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-gHBsuD&usg=ALkJrhinHumhGbKHk5f48GiXm5gslJvS-w)

Salida:

Primer elemento en intList: 7  
1st Even Element en intList: 10  
Primer elemento en strList:  
Primer elemento en emptyList:

Tenga cuidado al especificar las condiciones en First () o FirstOrDefault (). First () lanzará una excepción si una colección no incluye ningún elemento que satisfaga la condición especificada o incluya un elemento nulo.

Si una colección incluye un elemento nulo, FirstOrDefault () arroja una excepción mientras se evalúa la condición especificada. El siguiente ejemplo demuestra esto.

Ejemplo: LINQ First () y FirstOrDefault () - C #

IList < int > intList = new List< int >() { 7, 10, 21, 30, 45, 50, 87 }; IList < string > strList = new List< string >() { null , "Two" , "Three" , "Four" , "Five" }; Console .WriteLine( "1st Element which is greater than 250 in intList: {0}" , intList.First( i > 250)); Console .WriteLine( "1st Even Element in intList: {0}" , strList.FirstOrDefault(s => s.Contains("T")));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-78VqFh&usg=ALkJrhgDxET8DlKwvyvLSlq2AJzoV1A3YA)

Salida:

Excepción de tiempo de ejecución: la secuencia no contiene ningún elemento coincidente

# Operadores de elementos: Last & LastOrDefault

| **Operadores de elementos** | **Descripción** |
| --- | --- |
| Último | Devuelve el último elemento de una colección, o el último elemento que satisface una condición |
| LastOrDefault | Devuelve el último elemento de una colección, o el último elemento que satisface una condición.Devuelve un valor predeterminado si no existe dicho elemento. |

Last y LastOrDefault tiene dos métodos de sobrecarga. Un método de sobrecarga no toma ningún parámetro de entrada y devuelve el último elemento de la colección. El segundo método de sobrecarga toma una expresión lambda para especificar una condición y devuelve el último elemento que satisface la condición especificada.

Métodos de sobrecarga de Last y LastOrDefault - C #

public static TSource Last <TSource> ( esta fuente IEnumerable <TSource>);

public static TSource Last <TSource> ( esta fuente IEnumerable <TSource>, Func <TSource, bool > predicado);

public static TSource LastOrDefault <TSource> ( esta fuente IEnumerable <TSource>);

public static TSource LastOrDefault <TSource> ( esta fuente IEnumerable <TSource>, Func <TSource, bool > predicado);

El método Last () devuelve el último elemento de una colección o el último elemento que satisface la condición especificada utilizando la expresión lambda o el delegado de Func. Si una colección determinada está vacía o no incluye ningún elemento que satisfaga la condición, arrojará la excepción InvalidOperation.

El método LastOrDefault () hace lo mismo que el método Last (). La única diferencia es que devuelve el valor predeterminado del tipo de datos de una colección si una colección está vacía o no encuentra ningún elemento que satisfaga la condición.

El siguiente ejemplo muestra el método Last ().

Ejemplo: LINQ Last () - C #

IList < int > intList = new List< int >() { 7, 10, 21, 30, 45, 50, 87 }; IList < string > strList = new List< string >() { null , "Two" , "Three" , "Four" , "Five" }; IList < string > emptyList = new List< string >(); Console .WriteLine( "Last Element in intList: {0}" , intList.Last()); Console .WriteLine( "Last Even Element in intList: {0}" , intList.Last(i => i % 2 == 0)); Console .WriteLine( "Last Element in strList: {0}" , strList.Last()); Console .WriteLine( "emptyList.Last() throws an InvalidOperationException" ); Console .WriteLine( "-------------------------------------------------------------" ); Console .WriteLine(emptyList.Last());

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-48tdlj&usg=ALkJrhgFp9-rH5NhGx5F6V9YBXwfw1Jq4w)

Salida:

Último elemento en intList: 87  
Último elemento par en intList: 50  
Último elemento en strList: Cinco  
emptyList.Last () arroja una excepción InvalidOperationException  
-------------------------------------------------- -----------  
Excepción de tiempo de ejecución: la secuencia no contiene elementos ...

El siguiente ejemplo muestra el método LastOrDefault ().

Ejemplo: LINQ LastOrDefault () - C #

IList < int > intList = new List< int >() { 7, 10, 21, 30, 45, 50, 87 }; IList < string > strList = new List< string >() { null , "Two" , "Three" , "Four" , "Five" }; IList < string > emptyList = new List< string >(); Console .WriteLine( "Last Element in intList: {0}" , intList.LastOrDefault()); Console .WriteLine( "Last Even Element in intList: {0}" , intList.LastOrDefault(i => i % 2 == 0)); Console .WriteLine( "Last Element in strList: {0}" , strList.LastOrDefault()); Console .WriteLine( "Last Element in emptyList: {0}" , emptyList.LastOrDefault());

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-k7QILG&usg=ALkJrhj7HSCGLyPrwBCHH__zehAz54eiBQ)

Salida:

Último elemento en intList: 7  
Último elemento par en intList: 10  
Último elemento en strList:  
Último elemento en emptyList:

Tenga cuidado al especificar la condición en Last () o LastOrDefault (). Last () lanzará una excepción si una colección no incluye ningún elemento que satisfaga la condición especificada o incluya un elemento nulo.

Si una colección incluye un elemento nulo, LastOrDefault () arroja una excepción mientras se evalúa la condición especificada. El siguiente ejemplo demuestra esto.

Ejemplo: LINQ Last () y LastOrDefault () - C #

IList < int > intList = new List< int >() { 7, 10, 21, 30, 45, 50, 87 }; IList < string > strList = new List< string >() { null , "Two" , "Three" , "Four" , "Five" }; Console .WriteLine( "Last Element which is greater than 250 in intList: {0}" , intList.Last(i => i > 250)); Console .WriteLine( "Last Even Element in intList: {0}" , strList.LastOrDefault(s => s.Contains("T")));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-gGiJx9&usg=ALkJrhjhJrrWKAzuXoXorMFql58oBorZTg)

Salida:

Excepción de tiempo de ejecución: la secuencia no contiene ningún elemento coincidente

# Operadores de elementos: Single & SingleOrDefault

| **Operadores de elementos** | **Descripción** |
| --- | --- |
| Soltero | Devuelve el único elemento de una colección, o el único elemento que satisface una condición. Si Single () no encontró elementos o más de un elemento en la colección, arroja InvalidOperationException. |
| SingleOrDefault | Lo mismo que Single, excepto que devuelve un valor predeterminado de un tipo genérico especificado, en lugar de arrojar una excepción si no se encuentra ningún elemento para la condición especificada. Sin embargo, arrojará InvalidOperationException si encontró más de un elemento para la condición especificada en la colección. |

Single y SingleOrDefault tienen dos métodos de sobrecarga. El primer método de sobrecarga no toma ningún parámetro de entrada y devuelve un único elemento en la colección. El segundo método de sobrecarga toma la expresión lambda como un predicado delegado que especifica la condición y devuelve un único elemento que satisface la condición especificada.

Métodos de sobrecarga de Single y SingleOrDefault - C #

public static TSource Single <TSource> ( esta fuente IEnumerable <TSource>);

public static TSource Single <TSource> ( esta fuente IEnumerable <TSource>, Func <TSource, bool > predicado);

public static TSource SingleOrDefault <TSource> ( esta fuente IEnumerable <TSource>);

public static TSource SingleOrDefault <TSource> ( esta fuente IEnumerable <TSource>, Func <TSource, bool > predicado);

Single () devuelve el único elemento de una colección, o el único elemento que satisface la condición especificada. Si una colección determinada no incluye elementos o más de un elemento, Single () arroja InvalidOperationException.

El método SingleOrDefault () hace lo mismo que el método Single (). La única diferencia es que devuelve el valor predeterminado del tipo de datos de una colección si una colección está vacía, incluye más de un elemento o no encuentra ningún elemento o más de un elemento para la condición especificada.

Ejemplo: Single in method syntax C #

IList < int > oneElementList = new List< int >() { 7 }; IList < int > intList = new List< int >() { 7, 10, 21, 30, 45, 50, 87 }; IList < string > strList = new List< string >() { null , "Two" , "Three" , "Four" , "Five" }; IList < string > emptyList = new List< string >(); Console .WriteLine( "The only element in oneElementList: {0}" , oneElementList.Single()); Console .WriteLine( "The only element in oneElementList: {0}" , oneElementList.SingleOrDefault()); Console .WriteLine( "Element in emptyList: {0}" , emptyList.SingleOrDefault()); Console .WriteLine( "The only element which is less than 10 in intList: {0}" , intList.Single(i => i < 10)); //Followings throw an exception //Console.WriteLine("The only Element in intList: {0}", intList.Single()); //Console.WriteLine("The only Element in intList: {0}", intList.SingleOrDefault()); //Console.WriteLine("The only Element in emptyList: {0}", emptyList.Single());

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-AWvJXU&usg=ALkJrhh8qeduKXojsj6637qj4cFMrlBg4w)

Salida:

El único elemento en oneElementList: 7  
El único elemento en oneElementList: 7  
Elemento en emptyList: 0  
El único elemento que es menos de 10 en la lista int: 7

El siguiente código de ejemplo arroja una excepción porque Single () o SingleOrDefault () devuelve ninguno o múltiples elementos para la condición especificada.

C #: Single () y SingleOrDefault ()

IList < int > oneElementList = new List< int >() { 7 }; IList < int > intList = new List< int >() { 7, 10, 21, 30, 45, 50, 87 }; IList < string > strList = new List< string >() { null , "Two" , "Three" , "Four" , "Five" }; IList < string > emptyList = new List< string >(); //following throws error because list contains more than one element which is less than 100 Console .WriteLine( "Element less than 100 in intList: {0}" , intList.Single(i => i < 100)); //following throws error because list contains more than one element which is less than 100 Console .WriteLine( "Element less than 100 in intList: {0}" , intList.SingleOrDefault(i => i < 100)); //following throws error because list contains more than one elements Console .WriteLine("The only Element in intList: {0}", intList.Single()); //following throws error because list contains more than one elements Console .WriteLine("The only Element in intList: {0}", intList.SingleOrDefault()); //following throws error because list does not contains any element Console .WriteLine("The only Element in emptyList: {0}", emptyList.Single());

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-H7j26O)



### Puntos para recordar :

1. Single () espera un solo elemento en la colección.
2. Single () arroja una excepción cuando no obtiene ningún elemento o más de un elemento en la colección.
3. Si se especifica una condición en Single () y el resultado no contiene ningún elemento o más de un elemento, arroja una excepción.
4. SingleOrDefault () devolverá el valor predeterminado de un tipo de datos de colección genérica si no hay elementos en una colección o para la condición especificada.
5. SingleOrDefault () lanzará una excepción si hay más de un elemento en una colección o para la condición especificada.

Aprende sobre SequenceEqual a continuación.

# Operador de igualdad: SequenceEqual

Solo hay un operador de igualdad: SequenceEqual. El método SequenceEqual comprueba si la cantidad de elementos, el valor de cada elemento y el orden de los elementos en dos colecciones son iguales o no.

Si la colección contiene elementos de tipos de datos primitivos, entonces compara los valores y el número de elementos, mientras que la colección con elementos de tipo complejo, verifica las referencias de los objetos. Entonces, si los objetos tienen la misma referencia, entonces se los considera iguales, de lo contrario no se los considera iguales.

El siguiente ejemplo demuestra el método SequenceEqual con la colección de tipos de datos primitivos.

Ejemplo: SequenceEqual en Method Syntax C #

IList < string > strList1 = new List < string >(){ "One" , "Two" , "Three" , "Four" , "Three" }; IList < string > strList2 = new List < string >(){ "One" , "Two" , "Three" , "Four" , "Three" }; bool isEqual = strList1.SequenceEqual(strList2); // returns true Console .WriteLine(isEqual);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-sKU5W2&usg=ALkJrhjbvHYxbsR3TWVGM83SBqJpJ40ItA)

Salida:

cierto

Si el orden de los elementos no es el mismo, el método SequenceEqual () devuelve falso.

Ejemplo: SequenceEqual en Method Syntax C #

IList < string > strList1 = new List < string >(){ "One" , "Two" , "Three" , "Four" , "Three" }; IList < string > strList2 = new List < string >(){ "Two" , "One" , "Three" , "Four" , "Three" }; bool isEqual = strList1.SequenceEqual(strList2); // returns false Console .WriteLine(isEqual);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-QSOVLl&usg=ALkJrhg8qsYVXeMsG_HKfq2f9pD8-HjcHA)

Salida:

falso

El método de extensión SequenceEqual comprueba las referencias de dos objetos para determinar si dos secuencias son iguales o no. Esto puede dar un resultado incorrecto. Considera el siguiente ejemplo:

Ejemplo: SequenceEqual en C #

Student std = new Student () { StudentID = 1, StudentName = "Bill" }; IList < Student > studentList1 = new List < Student >(){ std }; IList < Student > studentList2 = new List < Student >(){ std }; bool isEqual = studentList1.SequenceEqual(studentList2); // returns true Student std1 = new Student () { StudentID = 1, StudentName = "Bill" }; Student std2 = new Student () { StudentID = 1, StudentName = "Bill" }; IList < Student > studentList3 = new List < Student >(){ std1}; IList < Student > studentList4 = new List < Student >(){ std2 }; isEqual = studentList3.SequenceEqual(studentList4); // returns false

Student std = new Student () { StudentID = 1, StudentName = "Bill" }; IList < Student > studentList1 = new List < Student >(){ std }; IList < Student > studentList2 = new List < Student >(){ std }; bool isEqual = studentList1.SequenceEqual(studentList2); // returns true Student std1 = new Student () { StudentID = 1, StudentName = "Bill" }; Student std2 = new Student () { StudentID = 1, StudentName = "Bill" }; IList < Student > studentList3 = new List < Student >(){ std1}; IList < Student > studentList4 = new List < Student >(){ std2 }; isEqual = studentList3.SequenceEqual(studentList4); // returns false

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-1rTcM3&usg=ALkJrhgaqCvDeiLKcz7kfQtwjQG6Po89sA)

En el ejemplo anterior, studentList1 y studentList2 contienen el mismo objeto de estudiante, std.Entonces studentList1.SequenceEqual(studentList2) devuelve verdadero. Pero, stdList1 y stdList2 contienen dos objetos de estudiante separados, std1 y std2. Entonces, stdList1.SequenceEqual(stdList2) devolverá false incluso si std1 y std2 contienen el mismo valor.

Para comparar los valores de dos colecciones de tipo complejo (tipo de referencia u objeto), debe implementar la interfaz IEqualityComperar <T> como se muestra a continuación.

Ejemplo: IEqualityComparer C #:

class StudentComparer : IEqualityComparer < Student > { public bool Equals( Student x, Student y) { if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower()) return true ; return false ; } public int GetHashCode( Student obj) { return obj.GetHashCode(); } }

Ahora, puede usar la clase anterior StudentComparer en el método de extensión SequenceEqual como un segundo parámetro para comparar los valores:

Ejemplo: Comparar elementos de tipo de objeto utilizando SequenceEqual C #

IList < Student > studentList1 = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; IList < Student > studentList2 = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; // following returns true bool isEqual = studentList1.SequenceEqual(studentList2, new StudentComparer ());

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-ZKhlYp&usg=ALkJrhhFM4bDrtv12byZREgKYGLchfhw4w)



### Puntos para recordar :

1. El método SequenceEqual compara el número de elementos y sus valores para tipos de datos primitivos.
2. El método SequenceEqual compara la referencia de objetos para tipos de datos complejos.
3. Use la clase IEqualityComparer para comparar dos colecciones de tipo complejo usando el método SequenceEqual.

Más información sobre el operador Concat.

# Operador de concatenación: Concat

El método Concat () agrega dos secuencias del mismo tipo y devuelve una nueva secuencia (colección).

Ejemplo: Concat en C #

IList < string > collection1 = new List < string >() { "One" , "Two" , "Three" }; IList < string > collection2 = new List < string >() { "Five" , "Six" }; var collection3 = collection1.Concat(collection2); foreach ( string str in collection3) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-8HPf46&usg=ALkJrhiL8otNS4TRJZTaBic089Q1dDCQ8w)

Salida:

Uno  
Dos  
Tres  
Cinco  
Seis

Ejemplo: Concat en C #

IList < int > collection1 = new List < int >() { 1, 2, 3 }; IList < int > collection2 = new List < int >() { 4, 5, 6 }; var collection3 = collection1.Concat(collection2); foreach ( int i in collection3) Console .WriteLine(i);

IList < int > collection1 = new List < int >() { 1, 2, 3 }; IList < int > collection2 = new List < int >() { 4, 5, 6 }; var collection3 = collection1.Concat(collection2); foreach ( int i in collection3) Console .WriteLine(i);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-HFzIdc&usg=ALkJrhivmqvtWmx4Wx6S-Dd2hvcG4wvvkA)

Salida:

1  
2  
3  
4  
5  
6

El operador Concat no es compatible con la sintaxis de consulta en C # o VB.Net.

# Operador de generación: DefaultIfEmpty:

El método DefaultIfEmpty () devuelve una nueva colección con el valor predeterminado si la colección dada en la que se invoca DefaultIfEmpty () está vacía.

Otro método de sobrecarga de DefaultIfEmpty () toma un parámetro de valor que debe reemplazarse con el valor predeterminado.

Considera el siguiente ejemplo.

Ejemplo: DefaultIfEmpty C #

IList < string > emptyList = new List < string >(); var newList1 = emptyList.DefaultIfEmpty(); var newList2 = emptyList.DefaultIfEmpty( "None" ); Console .WriteLine( "Count: {0}" , newList1.Count()); Console .WriteLine( "Value: {0}" , newList1.ElementAt(0)); Console .WriteLine( "Count: {0}" , newList2.Count()); Console .WriteLine( "Value: {0}" , newList2.ElementAt(0));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-K1oBnM&usg=ALkJrhhUxZzoPamUYexoEWQ457cT75Vhwg)

Salida:

Cuenta: 1  
Valor:  
Cuenta: 1  
Valor: Ninguno

En el ejemplo anterior, emptyList.DefaultIfEmpty() devuelve una nueva colección de cadenas con un elemento cuyo valor es nulo porque null es un valor predeterminado de cadena. Otro método emptyList.DefaultIfEmpty("None") devuelve una colección de cadenas con un elemento cuyo valor es "Ninguno" en lugar de nulo.

El siguiente ejemplo muestra la invocación de DefaultIfEmpty en la colección int.

Ejemplo: DefaultIfEmpty C #

IList < int > emptyList = new List < int >(); var newList1 = emptyList.DefaultIfEmpty(); var newList2 = emptyList.DefaultIfEmpty(100); Console .WriteLine( "Count: {0}" , newList1.Count()); Console .WriteLine( "Value: {0}" , newList1.ElementAt(0)); Console .WriteLine( "Count: {0}" , newList2.Count()); Console .WriteLine( "Value: {0}" , newList2.ElementAt(0));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-hOKOCu&usg=ALkJrhjdngaBDdm4gRrKXgCYBUCYLXTUSg)

Salida:

Cuenta: 1  
Valor: 0  
Cuenta: 1  
Valor: 100

El siguiente ejemplo muestra el método DefaultIfEmpty () en la recopilación de tipos complejos.

Ejemplo: DefaultIfEmpty C #:

IList < Student > emptyStudentList = new List < Student >(); var newStudentList1 = studentList.DefaultIfEmpty( new Student ()); var newStudentList2 = studentList.DefaultIfEmpty( new Student (){ StudentID = 0, StudentName = "" }); Console .WriteLine(" Count: {0} ", newStudentList1.Count()); Console .WriteLine(" Student ID: {0} ", newStudentList1.ElementAt(0)); Console .WriteLine(" Count: {0} ", newStudentList2.Count()); Console .WriteLine(" Student ID: {0} ", newStudentList2.ElementAt(0).StudentID);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-dkKCCx&usg=ALkJrhgfyG-wh3dRty8EtLVUwsfZqxmKIA)

Salida:

Cuenta: 1  
Identificación del Estudiante:  
Cuenta: 1  
Identificación del estudiante: 0

# Operadores de generación: vacío, rango, repetición

LINQ incluye los operadores de generación DefaultIfEmpty, Empty, Range & Repeat. Los métodos Empty, Range & Repeat no son métodos de extensión para IEnumerable o IQueryable, pero son simplemente métodos estáticos definidos en una clase estática Enumerable.

| **Método** | **Descripción** |
| --- | --- |
| Vacío | Devuelve una colección vacía |
| Distancia | Genera una colección de tipo IEnumerable <T> con un número especificado de elementos con valores secuenciales, comenzando desde el primer elemento. |
| Repetir | Genera una colección de tipo IEnumerable <T> con una cantidad específica de elementos y cada elemento contiene el mismo valor especificado. |

## Vacío:

El método Empty () no es un método de extensión de IEnumerable o IQueryable como otros métodos LINQ. Es un método estático incluido en la clase estática Enumerable. Por lo tanto, puede llamarlo de la misma manera que otros métodos estáticos como Enumerable.Empty <TResult> ().El método Empty () devuelve una colección vacía de un tipo especificado como se muestra a continuación.

Ejemplo: Enumerable.Empty ()

var emptyCollection1 = Enumerable .Empty< string >(); var emptyCollection2 = Enumerable .Empty< Student >(); Console .WriteLine(" Count: {0} ", emptyCollection1.Count()); Console .WriteLine(" Type: {0} ", emptyCollection1.GetType().Name ); Console .WriteLine(" Count: {0} ",emptyCollection2.Count()); Console .WriteLine(" Type: {0} ", emptyCollection2.GetType().Name );

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-7KN2GL&usg=ALkJrhhlNfzBIoQF-4YsrbCFaLixH1YbxQ)

Salida:

Tipo: Cadena []  
Cuenta: 0  
Tipo: Estudiante []  
Cuenta: 0

## Distancia:

El método Range () devuelve una colección de tipo IEnumerable <T> con un número especificado de elementos y valores secuenciales a partir del primer elemento.

Ejemplo: Enumerable.Range ()

var intCollection = Enumerable.Range(10, 10); Console .WriteLine(" Total Count: {0} ", intCollection.Count()); for ( int i = 0; i < intCollection.Count(); i++) Console .WriteLine( "Value at index {0} : {1}" , i, intCollection.ElementAt(i));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-kuwXq8&usg=ALkJrhgRe7APjGQx3oalqWzPCxGIMEg1Eg)

Salida:

Recuento total: 10  
Valor en el índice 0: 10  
Valor en el índice 1: 11  
Valor en el índice 2: 12  
Valor en el índice 3: 13  
Valor en el índice 4: 14  
Valor en el índice 5: 15  
Valor en el índice 6: 16  
Valor en el índice 7: 17  
Valor en el índice 8: 18  
Valor en el índice 9: 19

En el ejemplo anterior, Enumerable.Range(10, 10) crea una colección con 10 elementos enteros con los valores secuenciales a partir de 10. El primer parámetro especifica el valor inicial de los elementos y el segundo parámetro especifica el número de elementos para crear.

## Repetir:

El método Repeat () genera una colección de tipo IEnumerable <T> con una cantidad específica de elementos y cada elemento contiene el mismo valor especificado.

Ejemplo: repetir

var intCollection = Enumerable.Repeat< int >(10, 10); Console .WriteLine(" Total Count: {0} ", intCollection.Count()); for ( int i = 0; i < intCollection.Count(); i++) Console .WriteLine( "Value at index {0} : {1}" , i, intCollection.ElementAt(i));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-5Ydi1v&usg=ALkJrhiUstPPHgT_k6kVsMkSbcKukjd93w)

Salida:

Recuento total: 10  
Valor en el índice 0: 10  
Valor en el índice 1: 10  
Valor en el índice 2: 10  
Valor en el índice 3: 10  
Valor en el índice 4: 10  
Valor en el índice 5: 10  
Valor en el índice 6: 10  
Valor en el índice 7: 10  
Valor en el índice 8: 10  
Valor en el índice 9: 10

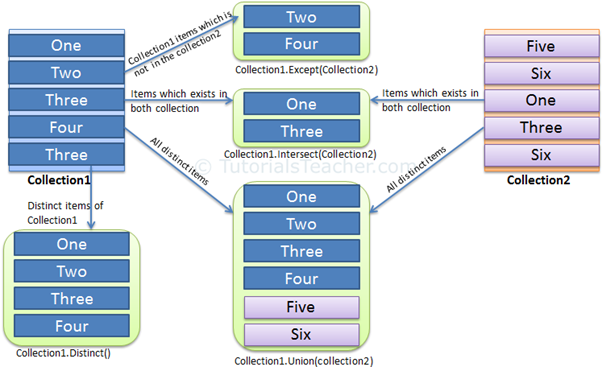
En el ejemplo anterior, Enumerable.Repeat< int >(10, 10) crea una colección con 100 elementos de tipo entero con el valor repetido de 10. El primer parámetro especifica los valores de todos los elementos y el segundo parámetro especifica el número de elementos para crear.

# Set operator: Distinct

La siguiente tabla enumera todos los operadores de Set disponibles en LINQ.

| **Establecer operadores** | **Uso** |
| --- | --- |
| Distinto | Devuelve valores distintos de una colección. |
| Excepto | Devuelve la diferencia entre dos secuencias, lo que significa los elementos de una colección que no aparecen en la segunda colección. |
| Intersecarse | Devuelve la intersección de dos secuencias, lo que significa elementos que aparecen en ambas colecciones. |
| Unión | Devuelve elementos únicos de dos secuencias, lo que significa elementos únicos que aparecen en cualquiera de las dos secuencias. |

La siguiente figura muestra cómo funciona cada conjunto de operadores en las colecciones:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-set-operators.png)Operadores del conjunto LINQ

## Distinto:

El método de extensión Distinct devuelve una nueva colección de elementos únicos de la colección dada.

Ejemplo: Distinct C #

IList < string > strList = new List < string >(){ "One" , "Two" , "Three" , "Two" , "Three" }; IList < int > intList = new List < int >(){ 1, 2, 3, 2, 4, 4, 3, 5 }; var distinctList1 = strList.Distinct(); foreach ( var str in distinctList1) Console .WriteLine(str); var distinctList2 = intList.Distinct(); foreach ( var i in distinctList2) Console .WriteLine(i);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-mq4rz2&usg=ALkJrhjfgAKR96yVVxHRXldHFa5-nq42mA)

Salida:

Uno  
Dos  
Tres  
1  
2  
3  
4  
5

El método de extensión Distinct no compara valores de objetos de tipo complejo.IEqualityComparer<T> implementar la IEqualityComparer<T> para comparar los valores de los tipos complejos. En el siguiente ejemplo, la clase IEqualityComparer<Student> implementa IEqualityComparer<Student> para comparar Student< objetos de Student< .

Ejemplo: Implementar IEqualityComparer en C #

public class Student { public int StudentID { get ; set ; } public string StudentName { get ; set ; } public int Age { get ; set ; } } class StudentComparer : IEqualityComparer < Student > { public bool Equals( Student x, Student y) { if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower()) return true ; return false ; } public int GetHashCode( Student obj) { return obj.StudentID.GetHashCode(); } }

Ahora puede pasar un objeto de la clase StudentComparer anterior en el método Distinct () como parámetro para comparar los objetos de Student como se muestra a continuación.

Ejemplo: Distinto en C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var distinctStudents = studentList.Distinct( new StudentComparer ()); foreach ( Student std in distinctStudents) Console .WriteLine(std.StudentName);

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-iwyGu1)

Salida:

John  
Steve  
Cuenta  
Ron

## Operador distintivo en la sintaxis de consulta:

El operador Distinct **no** es **compatible** con la sintaxis de C # Query. Sin embargo, puede usar el método Distinct de variable de consulta o envolver toda la consulta entre corchetes y luego llamar a Distinct ().

Utilice la palabra clave Distinct en la sintaxis de consulta de VB.Net:

Ejemplo: Distinto en la sintaxis de consulta VB.Net

Dim strList = New List (Of string ) From { "One" , "Three" , "Two" , "Two" , "One" } Dim distinctStr = From s In strList \_ Select s Distinct

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-ILlE3T)

# Establecer operador: Excepto

El método Except () requiere dos colecciones. Devuelve una nueva colección con elementos de la primera colección que no existen en la segunda colección (colección de parámetros).

Ejemplo: Excepto en la sintaxis del método C #

IList < string > strList1 = new List < string >(){ "One" , "Two" , "Three" , "Four" , "Five" }; IList < string > strList2 = new List < string >(){ "Four" , "Five" , "Six" , "Seven" , "Eight" }; var result = strList1.Except(strList2); foreach ( string str in result) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-QoyS7Z&usg=ALkJrhgVgn_k3lvLVj9X1rHu0aA3qMm6Eg)

Salida:

Uno  
Dos  
Tres

Excepto que el método de extensión no devuelve el resultado correcto para la colección de tipos complejos. Debe implementar la interfaz IEqualityComparer para obtener el resultado correcto del método Except.

Implemente la interfaz IEqualityComparer para la clase de Estudiante como se muestra a continuación:

Ejemplo: IEqualityComparer con el método Except C #

public class Student { public int StudentID { get ; set ; } public string StudentName { get ; set ; } public int Age { get ; set ; } } class StudentComparer : IEqualityComparer < Student > { public bool Equals( Student x, Student y) { if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower()) return true ; return false ; } public int GetHashCode( Student obj) { return obj.StudentID.GetHashCode(); } }

Ahora, puede pasar la clase StudentComparer en Excepto el método de extensión para obtener el resultado correcto:

Ejemplo: Excepto () con el tipo de objeto C #

IList < Student > studentList1 = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; IList < Student > studentList2 = new List < Student >() { new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var resultedCol = studentList1.Except(studentList2, new StudentComparer ()); foreach ( Student std in resultedCol) Console .WriteLine(std.StudentName);

IList < Student > studentList1 = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; IList < Student > studentList2 = new List < Student >() { new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var resultedCol = studentList1.Except(studentList2, new StudentComparer ()); foreach ( Student std in resultedCol) Console .WriteLine(std.StudentName);

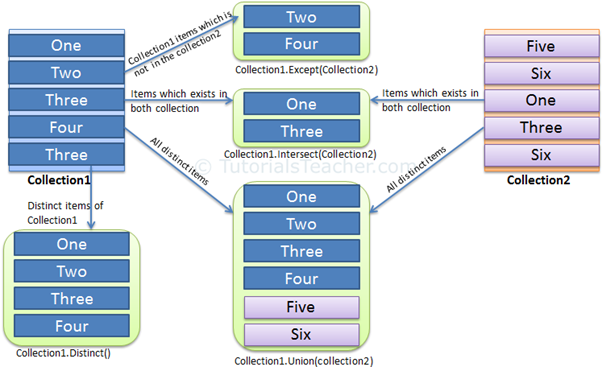
[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-uYqWlI&usg=ALkJrhirhWHjLvkRTIIve0EV0dPjVRMosw)

Salida:

John  
Steve

El operador Except **no** es **compatible** con la sintaxis C # & VB.Net Query. Sin embargo, puede usar el método Distinct en la variable de consulta o envolver toda la consulta entre corchetes y luego llamar a Except ().

La siguiente figura muestra cómo funciona cada conjunto de operadores en las colecciones:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-set-operators.png)Operadores del conjunto LINQ

# Establecer operador: intersecar

El método de extensión Intersect requiere dos colecciones. Devuelve una nueva colección que incluye elementos comunes que existen tanto en la colección. Considera el siguiente ejemplo.

Ejemplo: Intersecar en la sintaxis del método C #

IList < string > strList1 = new List < string >() { "One" , "Two" , "Three" , "Four" , "Five" }; IList < string > strList2 = new List < string >() { "Four" , "Five" , "Six" , "Seven" , "Eight" }; var result = strList1.Intersect(strList2); foreach ( string str in result) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-yte2C2&usg=ALkJrhhvLyeY-fhtK3dfBD03iMKmJml8Sw)

Salida:

Las cuatro  
Cinco

El método de extensión Intersecar no devuelve el resultado correcto para la colección de tipos complejos. Debe implementar la interfaz IEqualityComparer para obtener el resultado correcto del método Intersect.

Implemente la interfaz IEqualityComparer para la clase de Estudiante como se muestra a continuación:

Ejemplo: use IEqualityComparer con Intersect en C #

public class Student { public int StudentID { get ; set ; } public string StudentName { get ; set ; } public int Age { get ; set ; } } class StudentComparer : IEqualityComparer < Student > { public bool Equals( Student x, Student y) { if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower()) return true ; return false ; } public int GetHashCode( Student obj) { return obj.StudentID.GetHashCode(); } }

Ahora, puede pasar la clase StudentComparer en el método de extensión Intersect para obtener el resultado correcto:

Ejemplo: operador de intersección C #

IList < Student > studentList1 = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; IList < Student > studentList2 = new List < Student >() { new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var resultedCol = studentList1.Intersect(studentList2, new StudentComparer ()); foreach ( Student std in resultedCol) Console .WriteLine(std.StudentName);

IList < Student > studentList1 = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; IList < Student > studentList2 = new List < Student >() { new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var resultedCol = studentList1.Intersect(studentList2, new StudentComparer ()); foreach ( Student std in resultedCol) Console .WriteLine(std.StudentName);

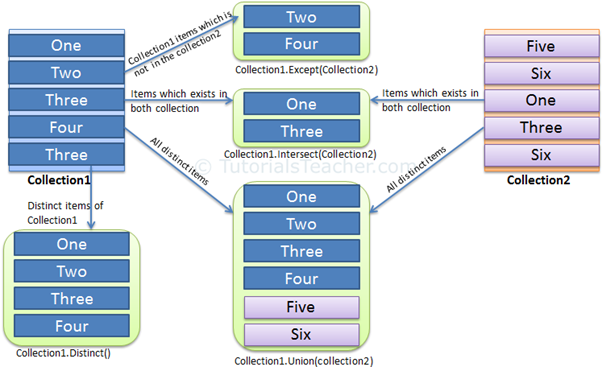
[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-rdgEDC&usg=ALkJrhhZcqhu9GCU5MZRhyWf8Ngdl020Zg)

Salida:

Cuenta  
Ron

El operador Intersecar **no** es **compatible** con la sintaxis C # & VB.Net Query. Sin embargo, puede usar el método Intersect en una variable de consulta o envolver toda la consulta entre corchetes y luego llamar a Intersect ().

La siguiente figura muestra cómo funciona cada conjunto de operadores en las colecciones:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-set-operators.png)Operadores del conjunto LINQ

# Establecer operador: Unión

El método de extensión de Unión requiere dos colecciones y devuelve una nueva colección que incluye elementos distintos de ambas colecciones. Considera el siguiente ejemplo.

Ejemplo: Unión () en C #

IList < string > strList1 = new List < string >() { "One" , "Two" , "three" , "Four" }; IList < string > strList2 = new List < string >() { "Two" , "THREE" , "Four" , "Five" }; var result = strList1.Union(strList2); foreach ( string str in result) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-ZhJcFJ&usg=ALkJrhhr5He6oxvVXxzWz9l3-hMq4NtoVA)

Salida:

Uno  
Dos  
Tres  
TRES  
Las cuatro  
Cinco

El método de extensión de la Unión no devuelve el resultado correcto para la recopilación de tipos complejos. Debe implementar la interfaz IEqualityComparer para obtener el resultado correcto del método Union.

Implemente la interfaz IEqualityComparer para la clase de Estudiante de la siguiente manera:

Ejemplo: operador de la Unión con IEqualityComparer:

public class Student { public int StudentID { get ; set ; } public string StudentName { get ; set ; } public int Age { get ; set ; } } class StudentComparer : IEqualityComparer < Student > { public bool Equals( Student x, Student y) { if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower()) return true ; return false ; } public int GetHashCode( Student obj) { return obj.StudentID.GetHashCode(); } }

Ahora puede pasar por encima de la clase StudentComparer en el método de extensión Union para obtener el resultado correcto:

Ejemplo: operador sindical C #

IList < Student > studentList1 = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; IList < Student > studentList2 = new List < Student >() { new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 19 } }; var resultedCol = studentList1.Union(studentList2, new StudentComparer ()); foreach ( Student std in resultedCol) Console .WriteLine(std.StudentName);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-swB0oA&usg=ALkJrhhUZBZfer774nTC1Pu8sYIMhzmuig)

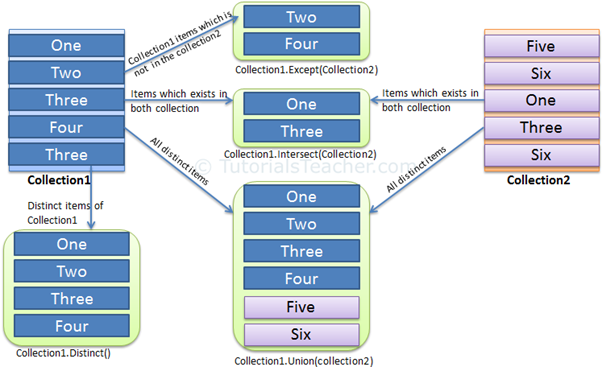
Salida:

John  
Steve  
Cuenta  
Ron

### Sintaxis de consulta:

El operador de la Unión **no** es **compatible** con la sintaxis C # & VB.Net Query. Sin embargo, puede usar el método Union en la variable de consulta o envolver la consulta completa en corchetes y luego llamar a Union ().

La siguiente figura muestra cómo funciona cada conjunto de operadores en las colecciones:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-set-operators.png)Operadores del conjunto LINQ

# Operadores de particionamiento: Skip & SkipWhile

Los operadores de particionamiento dividen la secuencia (colección) en dos partes y devuelven una de las partes.

| **Método** | **Descripción** |
| --- | --- |
| Omitir | Salta elementos hasta una posición específica comenzando desde el primer elemento en una secuencia. |
| SkipWhile | Salta elementos basados ​​en una condición hasta que un elemento no satisfaga la condición. Si el primer elemento en sí no satisface la condición, entonces omite 0 elementos y devuelve todos los elementos en la secuencia. |
| Tomar | Toma elementos hasta una posición específica comenzando desde el primer elemento en una secuencia. |
| TakeWhile | Devuelve elementos del primer elemento hasta que un elemento no satisfaga la condición. Si el primer elemento en sí no satisface la condición, devuelve una colección vacía. |

## Omitir:

El método Skip () omite el número de elemento especificado a partir del primer elemento y devuelve el resto de los elementos.

Ejemplo: Omitir () - C #

IList < string > strList = new List < string >(){ "One" , "Two" , "Three" , "Four" , "Five" }; var newList = strList.Skip(2); foreach ( var str in newList) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-UolWDJ&usg=ALkJrhi5VM3yi5UOEzlBPg33Xkpj9-ZY6g)

Salida:

Tres  
Las cuatro  
Cinco

## Omitir operador en la sintaxis de consulta:

El operador Omitir y OmitirModa **no** es **compatible con la sintaxis de consulta de C #** . Sin embargo, puede usar el método Saltar / Saltar en el camino en una variable de consulta o envolver la consulta completa entre corchetes y luego llamar a Saltar / SaltarMientras.

El siguiente ejemplo muestra el operador omitir en la sintaxis de la consulta - VB.NET

Ejemplo: Omitir operador en VB.Net

Dim skipResult = From s In studentList Skip 3 Select s

## SkipWhile:

Como su nombre indica, el método de extensión SkipWhile () en LINQ omite elementos en la colección hasta que la condición especificada sea verdadera. Devuelve una nueva colección que incluye todos los elementos restantes una vez que la condición especificada se vuelve falsa para cualquier elemento.

El método SkipWhile () tiene dos métodos de sobrecarga. Un método acepta el predicado de Func<TSource, bool> type y otro método de sobrecarga acepta el predicado Func<TSource, int, bool>type que pasa el índice de un elemento.

En el siguiente ejemplo, el método SkipWhile () omite todos los elementos hasta que encuentra una cadena cuya longitud es igual o superior a 4 caracteres.

Ejemplo: SkipWhile en C #

IList < string > strList = new List < string >() { "One" , "Two" , "Three" , "Four" , "Five" , "Six" }; var resultList = strList.SkipWhile(s => s.Length < 4); foreach ( string str in resultList) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-BDJxRr&usg=ALkJrhhTFp06KCT19nu6y0c7T_baKlqWaA)

Salida:

Tres  
Las cuatro  
Cinco  
Seis

En el ejemplo anterior, SkipWhile () omite los dos primeros elementos porque su longitud es menor que 3 y encuentra el tercer elemento cuya longitud es igual o mayor que 4. Una vez que encuentra cualquier elemento cuya longitud sea igual o mayor a 4 caracteres, no lo hará omita cualquier otro elemento, incluso si tienen menos de 4 caracteres.

Ahora, considere el siguiente ejemplo donde SkipWhile () no omite ningún elemento porque la condición especificada es falsa para el primer elemento.

Ejemplo: SkipWhile en C #

IList < string > strList = new List < string >() { "Three" , "One" , "Two" , "Four" , "Five" , "Six" }; var resultList = strList.SkipWhile(s => s.Length < 4); foreach ( string str in resultList) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-6KqbOa&usg=ALkJrhgKOWACYhAgUWSRkG7XrfBEiTcegQ)

Salida:

Tres  
Uno  
Dos  
Las cuatro  
Cinco  
Seis

La segunda sobrecarga de SkipWhile pasa un índice de cada elemento. Considera el siguiente ejemplo.

Ejemplo: SkipWhile con índice en C #

IList < string > strList = new List < string >() { "One" , "Two" , "Three" , "Four" , "Five" , "Six" }; var result = strList.SkipWhile((s, i) => s.Length > i); foreach ( string str in result) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-98qlw8&usg=ALkJrhils3Yms52JB0Xhv_bK1pWUIX98cw)

Salida:

Cinco  
Seis

En el ejemplo anterior, la expresión lambda incluye elementos e índices de un elemento como parámetro. Salta todos los elementos hasta que la longitud de un elemento de cadena sea mayor que su índice.

## Operador Skip / SkipWhile en Query Sintaxis:

Omitir y Omitir si el operador **NO** es **compatible con la sintaxis de consulta de C #** . Sin embargo, puede usar el método Omitir / Saltar en una variable de consulta o envolver toda la consulta entre corchetes y luego llamar a Omitir / OmitirMientras ().

Ejemplo: Método SkipWhile en VB.Net

Dim strList = New List (Of string ) From { "One" , "Two" , "Three" , "Four" , "Five" , "Six" } Dim skipWhileResult = From s In studentList Skip While s.Length < 4 Select s

Salida:

Tres  
Las cuatro  
Cinco  
Seis

# Operadores de particionamiento: Take & TakeWhile

Los operadores de particionamiento dividen la secuencia (colección) en dos partes y devuelve una de las partes.

| **Método** | **Descripción** |
| --- | --- |
| Omitir | Salta elementos hasta una posición específica comenzando desde el primer elemento en una secuencia. |
| SkipWhile | Salta elementos basados ​​en una condición hasta que un elemento no satisfaga la condición. Si el primer elemento en sí no satisface la condición, entonces omite 0 elementos y devuelve todos los elementos en la secuencia. |
| Tomar | Toma elementos hasta una posición específica comenzando desde el primer elemento en una secuencia. |
| TakeWhile | Devuelve elementos de la colección dada hasta que la condición especificada sea verdadera. Si el primer elemento en sí no satisface la condición, devuelve una colección vacía. |

## Tomar:

El método de extensión Take () devuelve el número especificado de elementos a partir del primer elemento.

Ejemplo: Take () en C #

IList < string > strList = new List < string >(){ "One" , "Two" , "Three" , "Four" , "Five" }; var newList = strList.Take(2); foreach ( var str in newList) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-RvBRFS&usg=ALkJrhg59ZEghSvHdAMfJDCs5Kba1h62Hg)

Salida:

Uno  
Dos

El operador Take and TakeWhile **no** es **compatible con la sintaxis de consulta de C #** . Sin embargo, puede usar el método Take / TakeWhile en la variable de consulta o envolver toda la consulta entre corchetes y luego llamar a Take / TakeWhile ().

Ejemplo: Tomar operador en sintaxis de consulta VB.Net

Dim takeResult = From s In studentList Take 3 Select s

## TakeWhile:

El método de extensión TakeShile () devuelve elementos de la colección dada hasta que la condición especificada sea verdadera. Si el primer elemento en sí no satisface la condición, devuelve una colección vacía.

El método TakeWhile tiene dos métodos de sobrecarga. Un método acepta el predicado de Func<TSource, bool> type y el otro método de sobrecarga acepta el predicado Func<TSource, int, bool> type que pasa el índice del elemento.

En el siguiente ejemplo, el método TakeWhile () devuelve una nueva colección que incluye todos los elementos hasta que encuentra una cadena cuya longitud es inferior a 4 caracteres.

Ejemplo: TakeWhile en C #

IList < string > strList = new List < string >() { "Three" , "Four" , "Five" , "Hundred" }; var result = strList.TakeWhile(s => s.Length > 4); foreach ( string str in result) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-V5sNbl&usg=ALkJrhgPIPVW39ba5uGz0zfIISts_Uml1g)

Salida:

Tres

En el ejemplo anterior, TakeWhile () incluye solo el primer elemento porque el segundo elemento de cadena no satisface la condición.

TakeWhile también pasa un índice del elemento actual en la función de predicado. El siguiente ejemplo del método TakeWhile toma elementos hasta que la longitud del elemento de cadena sea mayor de lo que es el índice

Ejemplo: TakeWhile en C #:

IList < string > strList = new List < string >() { "One" , "Two" , "Three" , "Four" , "Five" , "Six" }; var resultList = strList.TakeWhile((s, i) => s.Length > i); foreach ( string str in resultList) Console .WriteLine(str);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-cl3peb&usg=ALkJrhguDvdTc_Lh5_5oB5bX1RE7wKkomQ)

Salida:

Uno  
Dos  
Tres  
Las cuatro

# Operadores de conversión:

Los operadores de conversión en LINQ son útiles para convertir el tipo de elementos en una secuencia (colección). Hay tres tipos de operadores de conversión: **como** operadores (AsEnumerable y AsQueryable), operadores **To** (ToArray, ToDictionary, ToList y ToLookup), y operadores de **Casting** (Cast y OfType).

La siguiente tabla enumera todos los operadores de conversión.

| **Método** | **Descripción** |
| --- | --- |
| AsEnumerable | Devuelve la secuencia de entrada como IEnumerable <t> |
| AsQueryable | Convierte IEnumerable a IQueryable, para simular un proveedor de consulta remota |
| Emitir | Oculta una colección no genérica en una colección genérica (IEnumerable to IEnumerable <T>) |
| [OfType](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/linq/linq-filtering-operators-oftype&usg=ALkJrhiLSjsl14MHvQihU78VWHpOlMYAzg) | Filtra una colección basada en un tipo especificado |
| ToArray | Convierte una colección en una matriz |
| ToDictionary | Coloca elementos en un diccionario basado en la función de selector de tecla |
| Listar | Convierte la colección a List |
| [Para buscar](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/linq/linq-grouping-operator-groupby-tolookup&usg=ALkJrhh_ZpD2Y2eY5fiwcqJgejJyB5ezlA) | Agrupa elementos en una búsqueda <TKey, TElement> |

## AsEnumerable y AsQueryable:

Los métodos AsEnumerable y AsQueryable crean o convierten un objeto fuente en IEnumerable <T> o IQueryable <T>, respectivamente.

Considere el siguiente ejemplo: (cortesía: [Jon Skeet](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://stackoverflow.com/a/9063184/861716&usg=ALkJrhh_3epXfWaPyCLAaunFtfEuqbu8gg) )

Ejemplo: operador AsEnumerable & AsQueryable en C #:

class Program { static void ReportTypeProperties<T>(T obj) { Console .WriteLine( "Compile-time type: {0}" , typeof (T).Name); Console .WriteLine( "Actual type: {0}" , obj.GetType().Name); } static void Main( string [] args) { Student [] studentArray = { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 31 } , }; ReportTypeProperties( studentArray); ReportTypeProperties(studentArray.AsEnumerable()); ReportTypeProperties(studentArray.AsQueryable()); } }

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-QF5vgi&usg=ALkJrhhzQBCj_4plJe9CEphe80m9KL2pGA)

Salida:

Tipo de tiempo de compilación: Estudiante []  
Tipo real: Estudiante []  
Tipo de tiempo de compilación: IEnumerable`1  
Tipo real: Estudiante []  
Tipo de tiempo de compilación: IQueryable`1  
Tipo real: EnumerableQuery`1

Como puede ver en el ejemplo anterior, los métodos AsEnumerable y AsQueryable convierten el tipo de tiempo de compilación a IEnumerable y IQueryable respectivamente

Visite [stackoverflow](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://stackoverflow.com/questions/17968469/whats-the-differences-between-tolist-asenumerable-asqueryable&usg=ALkJrhiQ2_JgHYCl35CRbcZQbuLjEuuWaQ) para obtener información detallada sobre el método AsEnumerable y AsQueryable.

## Emitir:

Cast hace lo mismo que AsEnumerable <T>. Fundió el objeto fuente en IEnumerable <T>.

Ejemplo: operador de cast en C #

class Program { static void ReportTypeProperties<T>(T obj) { Console .WriteLine( "Compile-time type: {0}" , typeof (T).Name); Console .WriteLine( "Actual type: {0}" , obj.GetType().Name); } static void Main( string [] args) { Student [] studentArray = { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 25 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 31 } , }; ReportTypeProperties( studentArray); ReportTypeProperties(studentArray.Cast< Student >()); } }

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-WdHqn0&usg=ALkJrhhjv_Y14JCrBOuK-7hPy9Q-PLHt-w)

Salida:

Tipo de tiempo de compilación: Estudiante []  
Tipo real: Estudiante []  
Tipo de tiempo de compilación: IEnumerable`1  
Tipo real: Estudiante []  
Tipo de tiempo de compilación: IEnumerable`1  
Tipo real: Estudiante []  
Tipo de tiempo de compilación: IEnumerable`1  
Tipo real: Estudiante []

studentArray.Cast< Student >() es lo mismo que ( IEnumerable < Student >)studentArray pero Cast <Student> () es más legible.

## Para los operadores: ToArray (), ToList (), ToDictionary ():

Como su nombre lo sugiere, el método ToArray (), ToList (), ToDictionary () convierte un objeto fuente en una matriz, Lista o Diccionario, respectivamente.

**Para los** operadores forzar la ejecución de la consulta. Obliga al proveedor de consulta remota a ejecutar una consulta y obtener el resultado de la fuente de datos subyacente, por ejemplo, la base de datos de SQL Server.

Ejemplo: ToArray & ToList en C #

IList < string > strList = new List< string >() { "One" , "Two" , "Three" , "Four" , "Three" }; string [] strArray = strList.ToArray< string >(); // converts List to Array IList < string > list = strArray.ToList< string >(); // converts array into list

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-R5fkLe&usg=ALkJrhhHSvwykQPfpu5vetYFgYb752TxVg)

ToDictionary: convierte una lista genérica en un diccionario genérico:

Ejemplo: ToDictionary en C #:

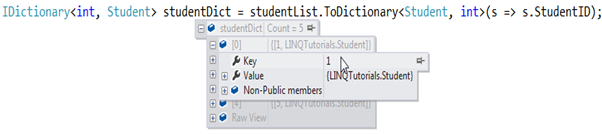
IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , age = 21 } }; //following converts list into dictionary where StudentId is a key IDictionary < int , Student > studentDict = studentList.ToDictionary< Student , int >(s => s.StudentID); foreach ( var key in studentDict.Keys) Console.WriteLine( "Key: {0}, Value: {1}" , key, (studentDict[key] as Student).StudentName);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-MgL8fL&usg=ALkJrhgKUwnl3r3xbKIav3j6n-j43cVdyA)

Salida:

Clave: 1, valor: John  
Clave: 2, valor: Steve  
Clave: 3, valor: factura  
Clave: 4, valor: Ram  
Clave: 5, Valor: Ron

La siguiente figura muestra cómo studentDict en el ejemplo anterior contiene un par clave-valor, donde key es un StudentID y el valor es Student object.

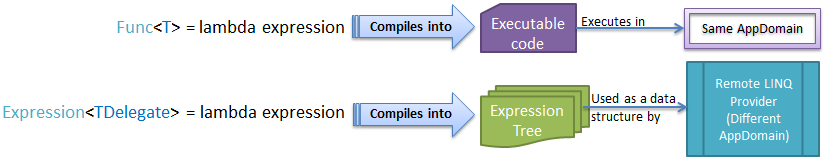
[](http://www.tutorialsteacher.com/Content/images/linq/linq-todictionary.png)Operador LINQ-ToDictionary

# Expresión en LINQ:

Hemos aprendido que la [Expresión de lambda](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/linq/linq-lambda-expression&usg=ALkJrhjSONdFXdmGfI9Lv4jWDSq4OPLAPA) se puede asignar a los delegados de Func o de tipo Acción para procesar las colecciones en memoria. El compilador .NET convierte la expresión lambda asignada a Func o delegado de tipo Acción en código ejecutable en tiempo de compilación.

LINQ presentó el nuevo tipo llamado [Expression](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://msdn.microsoft.com/en-us/library/bb335710(v%3Dvs.110).aspx&usg=ALkJrhjToFyTnmNp9foWV5f5-SEGnmnH0Q) que representa la expresión lambda fuertemente tipada. Significa que la expresión lambda también se puede asignar al tipo Expression <TDelegate>. El compilador .NET convierte la expresión lambda que está asignada a Expression <TDelegate> en un [árbol de expresiones en](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/linq/expression-tree&usg=ALkJrhiICGT5EqDYrUWGgCG41Jl3qLhdNA) lugar de código ejecutable. Los proveedores de consultas de LINQ remotos utilizan este árbol de expresiones como una estructura de datos para crear una consulta en tiempo de ejecución (como LINQ-to-SQL, EntityFramework o cualquier otro proveedor de consultas LINQ que implemente la interfaz IQueryable <T>).

La siguiente figura ilustra las diferencias cuando la expresión lambda asignada a Func o Action delega y la expresión en LINQ.

[](http://www.tutorialsteacher.com/Content/images/linq/expression.png)Expresión y Func

Aprenderemos el árbol de Expresión en la siguiente sección, pero primero, veamos cómo definir e invocar una Expresión.

## Definir una expresión:

Tome la referencia del espacio de nombres *System.Linq.Expressions* y use una clase Expression <TDelegate> para definir una Expresión. Expression <TDelegate> requiere un tipo de delegado Func o Action.

Por ejemplo, puede asignar una expresión lambda a la variable isTeenAger del delegado de tipo Func, como se muestra a continuación:

Ejemplo: defina el delegado Func para una expresión en C #

public class Student { public int StudentID { get ; set ; } public string StudentName { get ; set ; } public int Age { get ; set ; } } Func < Student , bool > isTeenAger = s => s.Age > 12 && s.Age < 20;

Ejemplo: defina el delegado de Func para una expresión en VB.Net

Dim isTeenAger As Func ( Of Student , Boolean ) = Function (s) s.Age > 12 And s.Age < 20

Y ahora, puede convertir el delegado de tipo Func anterior en una expresión envolviendo el delegado Func con Expresson, como se muestra a continuación:

Ejemplo: Definir expresión en C #

Expression < Func < Student , bool >> isTeenAgerExpr = s => s.age > 12 && s.age < 20;

Ejemplo: definir expresión en VB.Net

Dim isTeenAgerExpr As Expression ( Func ( Of Student , Boolean )) = Function (s) s.Age > 12 And s.Age < 20

de la misma manera, también puede ajustar un delegado de tipo Acción <t> con Expresión si no devuelve un valor del delegado.

Ejemplo: Definir expresión en C #

Expression < Action < Student >> printStudentName = s => Console .WriteLine(s.StudentName);

Ejemplo: definir expresión en VB.Net

Dim printStudentName As Expression ( Action ( Of Student ) = Function (s) Console .WriteLine(s.StudentName);

Por lo tanto, puede definir el tipo Expression <TDelegate>. Ahora, veamos cómo invocar a delegado envuelto por un Expression <TDelegate>.

## Invocar una expresión:

Puede invocar al delegado envuelto por una Expresión del mismo modo que un delegado, pero primero debe compilarlo utilizando el método Compilar (). Compile () devuelve delegateof **Func** o **Action** type para que pueda invocarlo como un delegado.

Ejemplo: Invocar expresión en C #

Expression < Func < Student , bool >> isTeenAgerExpr = s => s.age > 12 && s.age < 20; //compile Expression using Compile method to invoke it as Delegate Func < Student , bool > isTeenAger = isTeenAgerExpr.Compile(); //Invoke bool result = isTeenAger( new Student (){ StudentID = 1, StudentName = "Steve", Age = 20});

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-vpCZfZ)

Ejemplo: Invocar expresión en VB.Net

Dim isTeenAgerExpr As Expression (Of Func (Of Student , Boolean )) = Function (s) s.Age > 12 And s.Age < 20 'compile Expression using Compile method to invoke it as Delegate Dim isTeenAger As Func (Of Student , Boolean ) = isTeenAgerExpr.Compile() Dim result = isTeenAger( New Student () With { .StudentID = 1, .StudentName = "Steve", .Age = 20})

[Intentalo](http://www.tutorialsteacher.com/codeeditor?cid=cs-hOPrST)

Aprende sobre el árbol de Expresión en detalle en la siguiente sección.

# Árbol de expresión:

Has aprendido sobre la [Expresión](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/linq/linq-expression&usg=ALkJrhgyp_H5HbHl6SQ1uGiFfG9AVestiQ) en la sección anterior. Ahora, vamos a aprender sobre el árbol Expresion aquí.

El árbol de expresiones como su nombre lo sugiere no es más que expresiones dispuestas en una estructura de datos similar a un árbol. Cada nodo en un árbol de expresiones es una expresión. Por ejemplo, un árbol de expresiones se puede usar para representar la fórmula matemática x <y donde x, <y y se representarán como una expresión y se organizarán en la estructura similar a un árbol.

El árbol de expresiones es una representación en memoria de una expresión lambda. Tiene los elementos reales de la consulta, no el resultado de la consulta.

El árbol de expresiones hace que la estructura de la expresión lambda sea transparente y explícita.Puede interactuar con los datos en el árbol de expresiones de la misma manera que con cualquier otra estructura de datos.

Por ejemplo, considere la siguiente expresión isTeenAgerExpr:

Ejemplo: Expresión en C #

Expression < Func < Student , bool >> isTeenAgerExpr = s => s.age > 12 && s.age < 20;

El compilador traducirá la expresión anterior en el siguiente árbol de expresiones:

Ejemplo: Árbol de expresiones en C #

Expression .Lambda< Func < Student , bool >>( Expression .AndAlso( Expression .GreaterThan( Expression .Property(pe, "Age" ), Expression .Constant(12, typeof ( int ))), Expression .LessThan( Expression .Property(pe, "Age" ), Expression .Constant(20, typeof ( int )))), new [] { pe });

También puedes construir un árbol de expresiones manualmente. Veamos cómo construir un árbol de expresiones para la siguiente expresión lambda simple:

Ejemplo: delegado de Func en C #:

Func < Student , bool > isAdult = s => s.age >= 18;

Este delegado de tipo Func se tratará como el siguiente método:

DO#:

public bool function( Student s) { return s.Age > 18; }

Para crear el árbol de expresiones, antes que nada, cree una expresión de parámetro donde Student es el tipo de parámetro y 's' es el nombre del parámetro como se muestra a continuación:

Paso 1: Crear expresión de parámetros en C #

ParameterExpression pe = Expression .Parameter( typeof ( Student ), "s" );

Ahora, use Expression.Property () para crear una expresión de origen donde s es el parámetro y Age es el nombre de propiedad de Student. (La **expresión** es una clase abstracta que contiene métodos estáticos de ayuda para crear el árbol de expresiones manualmente).

Paso 2: Crea una expresión de propiedad en C #

MemberExpression me = Expression .Property(pe, "Age" );

Ahora, crea una expresión constante para 18:

Paso 3: crea una expresión constante en C #

ConstantExpression constant = Expression .Constant(18, typeof ( int ));

Hasta ahora, hemos construido árboles de expresión para la edad del miembro (expresión del miembro) y 18 (expresión constante). Ahora necesitamos verificar si una expresión miembro es mayor que una expresión constante o no. Para eso, use el método Expression.GreaterThanOrEqual () y pase la expresión miembro y la expresión constante como parámetros:

Paso 4: crea una expresión binaria en C #

BinaryExpression body = Expression .GreaterThanOrEqual(me, constant);

Por lo tanto, hemos construido un árbol de expresiones para un cuerpo de expresión lambda en el origen> = 18. Ahora necesitamos unir el parámetro y las expresiones del cuerpo. Use Expression.Lambda (cuerpo, matriz de parámetros) para unir el cuerpo y la parte del parámetro de la expresión lambda s => s.age> = 18:

Paso 5: Crea una expresión Lambda en C #

var isAdultExprTree = Expression .Lambda< Func < Student , bool >>(body, new [] { pe });

De esta forma puedes construir un árbol de expresiones para delegados de Func simples con una expresión lambda.

Ejemplo: Árbol de expresiones en C #

ParameterExpression pe = Expression .Parameter( typeof ( Student ), "s" ); MemberExpression me = Expression .Property(pe, "Age" ); ConstantExpression constant = Expression .Constant(18, typeof ( int )); BinaryExpression body = Expression .GreaterThanOrEqual(me, constant); var ExpressionTree = Expression .Lambda< Func < Student , bool >>(body, new [] { pe }); Console.WriteLine( "Expression Tree: {0}" , ExpressionTree); Console.WriteLine( "Expression Tree Body: {0}" , ExpressionTree.Body); Console.WriteLine( "Number of Parameters in Expression Tree: {0}" , ExpressionTree.Parameters.Count); Console.WriteLine( "Parameters in Expression Tree: {0}" , ExpressionTree.Parameters[0]);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-XtoEjK&usg=ALkJrhik5QH1aKrba3Z2NfOh2GjJzjz2CA)

Ejemplo: Expression Tree en VB.Net

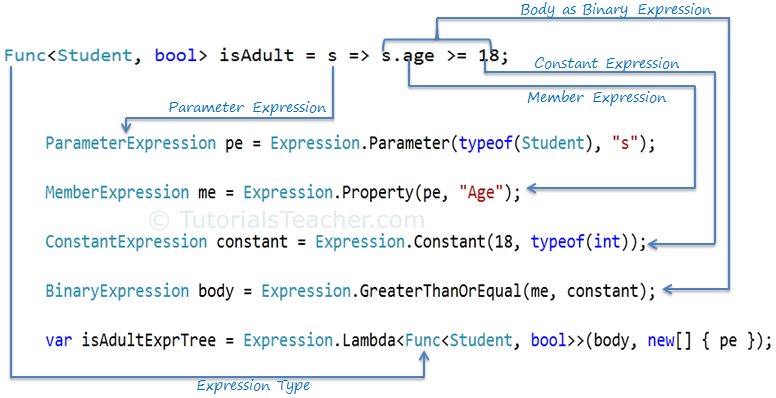
Dim pe As ParameterExpression = Expression.Parameter( GetType (Student), "s" ) Dim mexp As MemberExpression = Expression. Property (pe, "Age" ) Dim constant As ConstantExpression = Expression.Constant(18, GetType ( Integer )) Dim body As BinaryExpression = Expression.GreaterThanOrEqual(mexp, constant) Dim ExpressionTree As Expression(Of Func(Of Student, Boolean )) = Expression.Lambda(Of Func(Of Student, Boolean ))(body, New ParameterExpression() {pe}) Console.WriteLine( "Expression Tree: {0}" , ExpressionTree) Console.WriteLine( "Expression Tree Body: {0}" , ExpressionTree.Body) Console.WriteLine( "Number of Parameters in Expression Tree: {0}" , ExpressionTree.Parameters.Count) Console.WriteLine( "Parameters in Expression Tree: {0}" , ExpressionTree.Parameters(0))

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-fPHd9u&usg=ALkJrhi7c-6KD9gQJQEekJ-abuZyygkHuQ)

Salida:

Árbol de expresiones: s => (s.Age> = 18)  
Cuerpo del árbol de expresiones: (s.Age> = 18)  
Número de parámetros en el árbol de expresiones: 1  
Parámetros en Expression Tree: s

La siguiente imagen ilustra todo el proceso de creación de un árbol de expresiones:

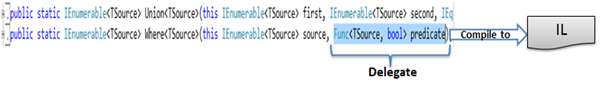
[](http://www.tutorialsteacher.com/Content/images/linq/linq-construct-expression-tree.png)Construir árbol de expresiones

## ¿Por qué Expression Tree?

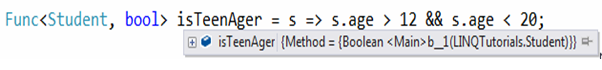
Hemos visto en la sección anterior que la expresión lambda asignada a ***Func <T>*** compila en código ejecutable y la expresión lambda asignada a ***Expression <TDelegate>*** escribe en árbol de Expresión.

El código ejecutable se ejecuta en el mismo dominio de aplicación para procesar la recopilación en memoria. Las clases estáticas enumerables contienen métodos de extensión para colecciones en memoria que implementan la **interfaz IEnumerable <T>** , por ejemplo, Lista <T>, Diccionario <T>, etc. Los métodos de Extensión en una clase Enumerable aceptan un parámetro de predicado de tipo **Func** delegado. Por ejemplo, el método de extensión **Where** acepta **Func <TSource, bool> predicate** . Luego lo compila en IL (lenguaje intermedio) para procesar las colecciones en memoria que están en el mismo dominio de la aplicación.

La siguiente imagen muestra el método de extensión Donde en la clase Enumerable incluye el delegado Func como parámetro:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-func-delegate-IEnumerable.png)Func delegate en Donde

**El** delegado de **Func** es un código ejecutable sin procesar, por lo que si depura el código, descubrirá que el delegado de **Func** se representará como código opaco. No puede ver sus parámetros, tipo de retorno y cuerpo:

[](http://www.tutorialsteacher.com/Content/images/linq/func-delegate-debug.png)Func delegate en modo de depuración

**El** delegado de **Func** es para colecciones en memoria porque se procesará en el mismo dominio de aplicación, pero ¿qué sucede con los proveedores de consultas LINQ remotos, como LINQ-to-SQL, EntityFramework u otros productos de terceros que proporcionan capacidades LINQ? ¿Cómo analizarían la expresión lambda que se compiló en el código ejecutable sin formato para conocer los parámetros, el tipo de retorno de la expresión lambda y la consulta del tiempo de ejecución de la compilación para seguir procesando? La respuesta es **Árbol de expresiones** .

Expression <TDelegate> se compila en una estructura de datos llamada árbol de expresiones.

Si depura el código, el delegado de expresión se representará como se muestra a continuación:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-expressiontree-debug.png)Árbol de expresiones en modo de depuración

Ahora puede ver la diferencia entre un delegado normal y una Expresión. Un árbol de expresión es transparente. Puede recuperar un parámetro, tipo de retorno y la información de expresión corporal de la expresión, como se muestra a continuación:

Ejemplo: Árbol de expresiones en C #

Expression < Func < Student , bool >> isTeenAgerExpr = s => s.Age > 12 && s.Age < 20; Console.WriteLine( "Expression: {0}" , isTeenAgerExpr ); Console.WriteLine( "Expression Type: {0}" , isTeenAgerExpr.NodeType); var parameters = isTeenAgerExpr.Parameters; foreach ( var param in parameters) { Console.WriteLine( "Parameter Name: {0}" , param.Name); Console.WriteLine( "Parameter Type: {0}" , param.Type.Name ); } var bodyExpr = isTeenAgerExpr.Body as BinaryExpression; Console.WriteLine( "Left side of body expression: {0}" , bodyExpr.Left); Console.WriteLine( "Binary Expression Type: {0}" , bodyExpr.NodeType); Console.WriteLine( "Right side of body expression: {0}" , bodyExpr.Right); Console.WriteLine( "Return Type: {0}" , isTeenAgerExpr.ReturnType);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-etakEj&usg=ALkJrhgM9T1EaxM-iaxMINhJM3EgucHMfg)

Salida:

Expresión: s => ((s.Age> 12) AndAlso (s.Age <20))  
Tipo de expresión: Lambda  
Nombre del parámetro: s  
Tipo de parámetro: Estudiante  
Lado izquierdo de la expresión corporal: (s.Age> 12)  
Tipo de Expresión Binaria: YAdemás  
Lado derecho de la expresión corporal: (s.Age <20)  
Tipo de devolución: System.Boolean

La consulta LINQ para LINQ-to-SQL o Entity Framework no se ejecuta en el mismo dominio de la aplicación. Por ejemplo, la siguiente consulta LINQ para Entity Framework nunca se ejecuta realmente dentro de su programa:

Ejemplo: consulta LINQ en C #

var query = from s in dbContext .Students where s.Age >= 18 select s;

Primero se traduce a una declaración de SQL y luego se ejecuta en el servidor de la base de datos.

El código encontrado en una expresión de consulta debe traducirse en una consulta SQL que se puede enviar a otro proceso como una cadena. Para LINQ-to-SQL o Entity Frameworks, ese proceso pasa a ser una base de datos del servidor SQL. Obviamente, será mucho más fácil traducir una estructura de datos como un árbol de expresiones en SQL que traducir un código IL o ejecutable sin formato en SQL porque, como ha visto, es fácil recuperar información de una expresión.

Los árboles de expresiones se crearon para la tarea de convertir código como una expresión de consulta en una cadena que se puede pasar a otro proceso y ejecutar allí.

La clase estática consultable incluye métodos de extensión que aceptan un parámetro predicado de tipo Expresión. Esta expresión de predicado se convertirá en un Árbol de expresiones y luego se pasará al proveedor de LINQ remoto como una estructura de datos para que el proveedor pueda generar una consulta adecuada desde el árbol de expresiones y ejecutar la consulta.

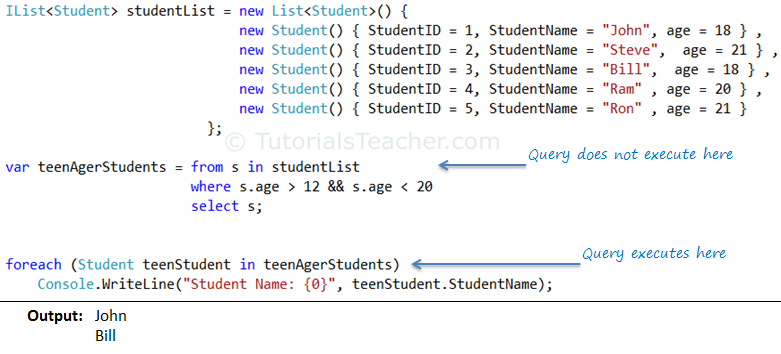
[](http://www.tutorialsteacher.com/Content/images/linq/linq-expressiontree-process.png)Proceso del árbol de expresiones

# Ejecución diferida de la consulta LINQ:

La ejecución diferida significa que la evaluación de una expresión se retrasa hasta que realmente se requiera su valor real. Mejora en gran medida el rendimiento al evitar una ejecución innecesaria.

La ejecución diferida es aplicable a cualquier colección en memoria así como a proveedores de LINQ remotos como LINQ-to-SQL, LINQ-to-Entities, LINQ-to-XML, etc.

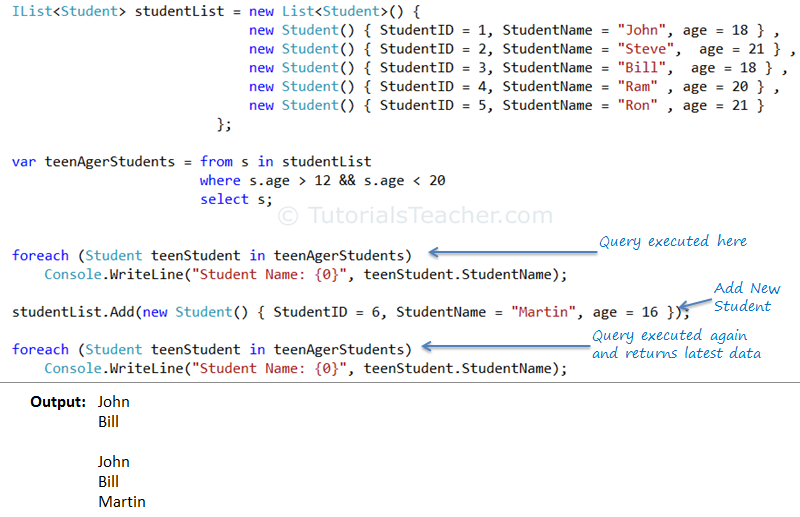
Comprendamos la ejecución diferida usando el siguiente ejemplo:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-deferred-execution-1.png)Ejecución diferida

En el ejemplo anterior, puede ver que la consulta se materializa y ejecuta cuando itera utilizando el bucle foreach. Esto se llama ejecución diferida. LINQ procesa la colección studentList cuando realmente accede a cada objeto de la colección y hace algo con él.

**La ejecución diferida devuelve los datos más recientes:**

Para verificar si la ejecución diferida devuelve los datos más recientes cada vez, agregue un estudiante adolescente más después del ciclo foreach y revise la lista de estudiantes adolescentes:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-deferred-execution-2.png)Ejecución diferida

Como puede ver, el segundo bucle foreach ejecuta nuevamente la consulta y devuelve los datos más recientes. La ejecución diferida vuelve a evaluar en cada ejecución; esto se llama **evaluación perezosa** . Esta es una de las principales ventajas de la ejecución diferida: siempre le brinda los últimos datos.

## Implementando la ejecución diferida:

Puede implementar la ejecución diferida para sus métodos de extensión personalizados para *IEnumerable* usando la palabra clave ***yield*** de C #.

Por ejemplo, puede implementar el método de extensión personalizado GetTeenAgerStudents para IEnumerable que devuelve una lista de todos los estudiantes que son adolescentes.

Ejemplo: Ejemplificación de la ejecución diferida en C #

public static class EnumerableExtensionMethods { public static IEnumerable < Student > GetTeenAgerStudents( this IEnumerable < Student > source) { foreach ( Student std in source) { Console .WriteLine( "Accessing student {0}" , std.StudentName); if (std.age > 12 && std.age < 20) yield return std; } } }

Tenga en cuenta que imprimimos el nombre del alumno en la consola cada vez que se llama a GetTeenAgerStudents ().

Ahora puede usar este método de extensión de la siguiente manera:

DO#:

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , age = 13 } , new Student () { StudentID = 2, StudentName = "Steve" , age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , age = 12 } , new Student () { StudentID = 5, StudentName = "Ron" , age = 21 } }; var teenAgerStudents = from s in studentList.GetTeenAgerStudents() select s; foreach ( Student teenStudent in teenAgerStudents) Console .WriteLine( "Student Name: {0}" , teenStudent.StudentName);

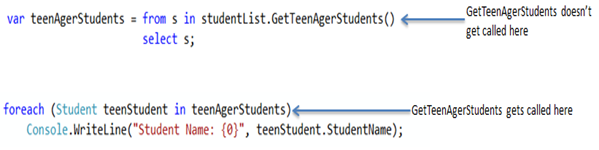
IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , age = 13 } , new Student () { StudentID = 2, StudentName = "Steve" , age = 15 } , new Student () { StudentID = 3, StudentName = "Bill" , age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , age = 12 } , new Student () { StudentID = 5, StudentName = "Ron" , age = 21 } }; var teenAgerStudents = from s in studentList.GetTeenAgerStudents() select s; foreach ( Student teenStudent in teenAgerStudents) Console .WriteLine( "Student Name: {0}" , teenStudent.StudentName);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-ER6Lim&usg=ALkJrhjj5eUbayHmYAhKcZD7t7Z-_JJEwA)

Salida:

Accediendo al estudiante John  
Nombre del estudiante: John  
Accediendo al estudiante Steve  
Nombre del estudiante: Steve  
Acceder a la cuenta estudiantil  
Nombre del estudiante: Bill  
Accediendo al estudiante Ram  
Accediendo al estudiante Ron

Como puede ver en la salida, se llama a GetTeenAgerStudents () cuando itera studentList usando el ciclo foreach.

[](http://www.tutorialsteacher.com/Content/images/linq/linq-deferred-execution-3.png)Ejecución diferida

De esta forma, puede crear métodos personalizados utilizando la palabra clave ***yield*** para obtener la ventaja de la ejecución diferida.

# Ejecución inmediata de la consulta LINQ:

La ejecución inmediata es el reverso de la ejecución diferida. Obliga a la consulta LINQ a ejecutar y obtiene el resultado de inmediato. Los [operadores de conversión](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/linq/linq-conversion-operators&usg=ALkJrhj3TPfT-U-bvOXm6_x_phd17pt6Gw) 'To' ejecutan la consulta dada y dan el resultado inmediatamente.

### Sintaxis del método:

En el siguiente ejemplo, el método de extensión ToList () ejecuta la consulta de inmediato y devuelve el resultado.

C #: Ejecución inmediata

IList < Student > teenAgerStudents = studentList.Where(s => s.age > 12 && s.age < 20).ToList();

VB.Net: Ejecución inmediata

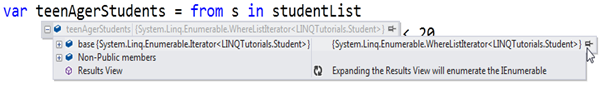
Dim teenAgerStudents As IList ( Of Student ) = studentList.Where( Function (s) s.Age > 12 And s.Age < 20).ToList()

### Sintaxis de consulta:

DO#:

var teenAgerStudents = from s in studentList where s.age > 12 && s.age < 20 select s;

La consulta anterior no se ejecutará de inmediato. No encontrará ningún resultado como se muestra a continuación:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-immedite-execution.png)Ejecución inmediata

Query Syntax no admite operadores 'To' pero puede usar ToList (), ToArray () o ToDictionary () para su ejecución inmediata como se muestra a continuación:

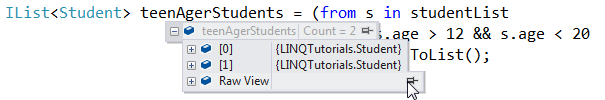
DO#:

IList < Student > teenAgerStudents = ( from s in studentList where s.age > 12 && s.age < 20 select s).ToList();

VB.Net:

Dim teenAgerStudents As IList ( Of Student ) = ( From s In studentList \_ Where s.Age > 12 And s.Age < 20 \_ Select s).ToList()

Puede ver el resultado en la colección teenAgerStudents, como se muestra a continuación:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-immedite-execution-2.png)Ejecución inmediata

### Otras lecturas:

* [El blog de Charlies](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://blogs.msdn.com/b/charlie/archive/2007/12/09/deferred-execution.aspx&usg=ALkJrhh_gxNcouRU9BXbSL_qto2-gllk6g)
* [MSDN](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://msdn.microsoft.com/en-us/library/bb943859.aspx&usg=ALkJrhh9iQ81ybjZ4rxrMnbNV9j43ZrAdw)

# dejar palabra clave:

La palabra clave 'let' es útil en la sintaxis de la consulta. Proyecta una nueva variable de rango, permite la reutilización de la expresión y hace que la consulta sea más legible.

Por ejemplo, puede comparar valores de cadena y seleccionar el valor de cadena en minúsculas como se muestra a continuación:

Ejemplo: dejar entrar consulta LINQ - C #

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 21 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 21 } }; var lowercaseStudentNames = from s in studentList where s.StudentName.ToLower().StartsWith( "r" ) select s.StudentName.ToLower();

Como puede ver, el método ToLower () se usa varias veces en la consulta anterior. El siguiente ejemplo usa 'let' para introducir la nueva variable 'lowquartseStudentName' que luego se usará en todas partes. Por lo tanto, **deje que la** palabra clave haga que la consulta sea más legible.

Ejemplo: let keyword in C #

var lowercaseStudentNames = from s in studentList let lowercaseStudentName = s.StudentName.ToLower() where lowercaseStudentName.StartsWith( "r" ) select lowercaseStudentName; foreach ( var name in lowercaseStudentNames) Console.WriteLine(name);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-5cCcTI&usg=ALkJrhgSjInqdkqvk8wCZAlACYXhMWvy3Q)

Ejemplo: let keyword en VB.Net

Dim lowercaseStudentNames = From s In studentList Let lowercaseStudentName = s.StudentName.ToLower() Where lowercaseStudentName.StartsWith( "r" ) Select lowercaseStudentName;

Salida:

RAM  
ron

# en palabra clave:

Ya hemos utilizado la palabra clave 'into' en la agrupación. También puede usar la palabra clave 'into' para continuar una consulta después de una cláusula de **selección** .

Ejemplo: en palabra clave en LINQ

var teenAgerStudents = from s in studentList where s.age > 12 && s.age < 20 select s into teenStudents where teenStudents.StudentName.StartsWith( "B" ) select teenStudents; var teenAgerStudents = from s in studentList where s.age > 12 && s.age < 20 select s into teenStudents where teenStudents.StudentName.StartsWith( "B" ) select teenStudents;

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-UxBg42&usg=ALkJrhgMlEip4R6SDg3wHqNlGY5mv9pnZw)

En la consulta anterior, la palabra clave 'into' introdujo una nueva variable *adolescente teenStudents* , por lo que la primera variable de rango **s** queda fuera del alcance. Puede escribir una consulta adicional después de la palabra clave into a través de una nueva variable de rango.

La palabra clave 'into' en VB.Net se usa para fines de agrupación.

Ejemplo: en palabra clave en LINQ VB.Net

Dim groupQuery = From s In studentList Group By s.Age Into Group

# Ejemplos de consultas LINQ:

En esta sección, aprenderá algunas consultas LINQ complejas. Utilizaremos la siguiente colección de Estudiantes y Estándar para nuestras consultas.

Colecciones de muestra:

IList < Student > studentList = new List < Student >() { new Student () { StudentID = 1, StudentName = "John" , Age = 18, StandardID = 1 } , new Student () { StudentID = 2, StudentName = "Steve" , Age = 21, StandardID = 1 } , new Student () { StudentID = 3, StudentName = "Bill" , Age = 18, StandardID = 2 } , new Student () { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID = 2 } , new Student () { StudentID = 5, StudentName = "Ron" , Age = 21 } }; IList < Standard > standardList = new List < Standard >() { new Standard (){ StandardID = 1, StandardName= "Standard 1" }, new Standard (){ StandardID = 2, StandardName= "Standard 2" }, new Standard (){ StandardID = 3, StandardName= "Standard 3" } };

## Operador de selección múltiple y dónde:

Ejemplo: operador de selección múltiple y dónde

var studentNames = studentList.Where(s => s.Age > 18) .Select(s => s) .Where(st => st.StandardID > 0) .Select(s => s.StudentName);

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-V4sRw3&usg=ALkJrhiW750q7tOLzObCNX34K9mjms7x-w)

Salida:

Steve  
RAM

La siguiente consulta devuelve Enumerable de objeto anónimo que solo tiene la propiedad StudentName:

Ejemplo: la consulta LINQ devuelve la colección de objetos anónimos

var teenStudentsName = from s in studentList where s.age > 12 && s.age < 20 select new { StudentName = s.StudentName }; teenStudentsName.ToList().ForEach(s => Console .WriteLine(s.StudentName));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-mSxFpU&usg=ALkJrhjFx6xqbPwZUG9ajanXgbiIRZnR8g)

Salida:

John  
Cuenta

## Agrupar por:

La siguiente consulta devuelve lista de grupos de estudiantes por StandardID:

Ejemplo: consulta LINQ GroupBy - C #

var studentsGroupByStandard = from s in studentList group s by s.StandardID into sg orderby sg.Key select new { sg.Key, sg }; foreach ( var group in studentsGroupByStandard) { Console .WriteLine( "StandardID {0}:" , group.Key); group.sg.ToList().ForEach(st => Console .WriteLine(st.StudentName )); }

var studentsGroupByStandard = from s in studentList group s by s.StandardID into sg orderby sg.Key select new { sg.Key, sg }; foreach ( var group in studentsGroupByStandard) { Console .WriteLine( "StandardID {0}:" , group.Key); group.sg.ToList().ForEach(st => Console .WriteLine(st.StudentName )); }

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-XN2vrG&usg=ALkJrhjFyG_MSy_NU1OlI__AEcfZaz2U9g)

Salida:

StandardID 0:  
Ron  
StandardID 1:  
John  
Steve  
StandardID 2:  
Cuenta  
RAM

El resultado incluye a Ron que no tiene ningún StandardID. Entonces Ron cae bajo el StandardID 0.

Para eliminar a un alumno que no tiene una ID estándar, utilice un operador where antes del operador del grupo:

Ejemplo: consulta LINQ GroupBy - C #

var studentsGroupByStandard = from s in studentList where s.StandardID > 0 group s by s.StandardID into sg orderby sg.Key select new { sg.Key, sg };

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-tEHfye&usg=ALkJrhhkdwRUwVm0Aw9jagdKjWlYZRCFHw)

Salida:

StandardID 1:  
John  
Steve  
StandardID 2:  
Cuenta  
RAM

## Izquierda combinación externa:

Use la combinación externa izquierda para mostrar a los estudiantes debajo de cada estándar.Muestre el nombre estándar incluso si no hay un estudiante asignado a ese estándar.

Ejemplo: LINQ Unión externa izquierda - C #

var studentsGroup = from stad in standardList join s in studentList on stad.StandardID equals s.StandardID into sg select new { StandardName = stad.StandardName, Students = sg }; foreach ( var group in studentsGroup) { Console .WriteLine(group.StandardName); group.Students.ToList().ForEach(st => Console .WriteLine(st.StudentName)); }

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-h0yBay&usg=ALkJrhjbhKJAxqsu5u8ad9nA9X9CiEPtcg)

Salida:

Estándar 1:  
John  
Steve  
Estándar 2:  
Cuenta  
RAM  
Estándar 3:

En el siguiente ejemplo de group by query, ordenamos el grupo y seleccionamos solo StudentName:

Ejemplo: LINQ Unión externa izquierda - C #

var studentsWithStandard = from stad in standardList join s in studentList on stad.StandardID equals s.StandardID into sg from std\_grp in sg orderby stad.StandardName, std\_grp.StudentName select new { StudentName = std\_grp.StudentName, StandardName = stad.StandardName }; foreach ( var group in studentsWithStandard) { Console .WriteLine( "{0} is in {1}" , group.StudentName, group.StandardName); }

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-JUmITE&usg=ALkJrhhCxDQxUT4lyNYWvuWIy_i_3gxV9w)

Salida:

John está en el estándar 1  
Steve está en el estándar 1  
Bill está en el estándar 2  
Ram está en el estándar 2

## Clasificación:

La siguiente consulta devuelve la lista de estudiantes por orden ascendente de StandardID y Age.

Ejemplo: clasificación

var sortedStudents = from s in studentList orderby s.StandardID, s.age select new { StudentName = s.StudentName, Age = s.age, StandardID = s.StandardID }; sortedStudents.ToList().ForEach(s => Console .WriteLine( "Student Name: {0}, Age: {1}, StandardID: {2}" , s.StudentName, s.Age , s.StandardID));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-GHHMp7&usg=ALkJrhie09tqdIHRmIhWGt_AuJcnzE0oLA)

Salida:

Nombre del estudiante: Ron, Edad: 21, StandardID: 0  
Nombre del estudiante: John, Edad: 18, StandardID: 1  
Nombre del estudiante: Steve, Edad: 21, StandardID: 1  
Nombre del alumno: Proyecto de ley, Edad: 18 años, Id. De norma: 2  
Nombre del estudiante: Ram, Edad: 20, StandardID: 2

## Unir internamente:

Ejemplo: unión interna LINQ - C #

var studentWithStandard = from s in studentList join stad in standardList on s.StandardID equals stad.StandardID select new { StudentName = s.StudentName, StandardName = stad.StandardName }; studentWithStandard.ToList().ForEach(s => Console.WriteLine( "{0} is in {1}" , s.StudentName, s.StandardName ));

var studentWithStandard = from s in studentList join stad in standardList on s.StandardID equals stad.StandardID select new { StudentName = s.StudentName, StandardName = stad.StandardName }; studentWithStandard.ToList().ForEach(s => Console.WriteLine( "{0} is in {1}" , s.StudentName, s.StandardName ));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-tD1rqa&usg=ALkJrhgKLLrcp6SInEf1wC984ST62eM9tw)

Salida:

John está en el estándar 1  
Steve está en el estándar 1  
Bill está en el estándar 2  
Ram está en el estándar 2

## Consulta anidada:

DO#:

var nestedQueries = from s in studentList where s.age > 18 && s.StandardID == ( from std in standardList where std.StandardName == "Standard 1" select std.StandardID).FirstOrDefault() select s; nestedQueries.ToList().ForEach(s => Console .WriteLine(s.StudentName));

[Intentalo](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=es&ie=UTF8&prev=_t&rurl=translate.google.com&sl=en&sp=nmt4&tl=es&u=http://www.tutorialsteacher.com/codeeditor%3Fcid%3Dcs-RG85mz&usg=ALkJrhi0ri8FkjWby5uf_PuHhwAV3g7EuQ)

Salida:

Steve