Seminario de Lenguajes de Programación
C# (Primera Parte)

Amalia Ibarra Rodríguez
Gabriela Martínez Giraldo
Sandra Martos Llanes
Paula Rodríguez Pérez

23 de marzo de 2020

Seminario 2 - LINQ

Language Integrated Query(LINQ) es un componente de la plataforma Microsoft.NET que añade capacidades de consulta a datos de manera nativa a los lenguajes de .NET. LINQ extiende el lenguaje mediante la adición de expresiones de consultas (query expressions), que son muy similares a las sentencias de SQL, y pueden ser usadas para convenientemente extraer y procesar datos de arrays, clases enumerables, documentos XML, bases de datos relacionales entre otros.

LINQ también define un conjunto de métodos (standard query operators o standard sequence operators), además de varias reglas usadas por el compilador para traducir queries de tipo fluent-style a expresiones usando estos métodos extensores y expresiones lambda, se unen a estas características del lenguaje las variables implícitamente tipadas, los inicializadores de objetos y los tipos anónimos.

Métodos Extensores

Los métodos extensores te permiten añadir métodos a tipos existentes sin tener que crear un nuevo tipo derivado, recompilar o modificar el tipo original. Los métodos extensores son un tipo especial de método estático, pero son llamados como si fueran métodos de instancia del tipo.

Ejemplo

El siguiente ejemplo implementa un método extensor llamado WordCount en la clase CustomExtensions.StringExtension. El método opera sobre la clase String, lo cual se especifica en el primer parámetro del método. El namespace CustomExtensions se importa en el namespace de la aplicación y el método se llama dentro del main.

```
// Extension methods must  be defined in a static class.
public static class StringExtension
{
        // This is the extension method.
        // The first parameter takes the " this"  modifier
        // and specifies the type for which the method is defined.
        public static int WordCount (this String str)
        {
                return str.Split(new char[ ]  {' ',  '. ', '?'}).
                   Length;
        }
}
static void Main(string[ ]  args)
{
        string s ="The quick brown fox jumped over the lazy dog.";
        // Call the method as if it were an
        // instance method on the type.Note that the first
        // parameter is not specified by the calling code.
        int i = s.WordCount();
        Console.WriteLine("Word count of sis {0}" ,  i);
}
```

IGrouping<TKey,TElement>Interfaz

Un IGrouping<TKey,TElement>es un IEnumerable<T>que además tiene una llave. La llave representa el atributo que es común para cada valor en el IGrouping<TKey,TElement>. Se puede acceder a los valores de un IGrouping<TKey,TElement>como mismo se accede a los elementos de un IEnumerable<T>, por ejemplo a través de un foreach. El método GroupBy retorna una secuencia de elementos de tipo IGrouping<TKey,TElement>. Ejemplos de lo anterior están disponibles en la carpeta Codes(Program.cs).

Pregunta 1.

Brinde una implementación eficiente y simple del siguiente método extensor y analice el costo operacional para el caso peor:

```
public static IEnumerable<IGrouping<TKey, TSource>>
    GroupBy<TSource, TKey> this IEnumerable<TSource> source
    , Func<TSource, TKey> keySelector)
```

Una aplicación útil de este método extensor sería:

```
var estudiantes = new List<Estudiante>();
// ...Algun codigo de inicializacion...
var Grupos = estudiantes.GroupBy(estudiante => estudiante.
    Grupo);
```

¿Se explotaría en su totalidad una implementación "Lazy" del `GroupBy`? ¿El costo de las operaciones para el caso peor es el mismo independientemente de si se hace un `Take(k)`?

Solución

Las implementaciones del método GroupBy y Take están disponibles en la carpeta Codes(Grouping.cs)

Costo Operacional

Sean N la cantidad de elementos en source y M la cantidad de llaves distintas ($M \leq N$), se realizan $O(N*(M+N))$ operaciones. Para el caso peor ($M = N$) el costo operacional es $O(N^2)$.

Si se hace un llamado a students.GroupBy(s =>s.Group).Take(k), como los grupos se devuelven por demanda sólo se forman los k primeros grupos, por tanto el costo operacional es $O(k^2)$.

Pregunta 2.

Reescriba el siguiente código de forma tal que siga manteniendo el `while(true)` pero que permita "parar" la ejecución del método para un momento dado:

```
static List<int> GetPrimes()
{
        var primes = new List<int>();
        int i = 1;
        while (true)
        {
                if (IsPrime(i)) primes.Add(i);
                i++;
        }
        return primes;
}
```

Solución:

```
 static IEnumerable<int> GetPrimesIterator()
 {
         var primes = new List<int>();
     int i = 1;
     while (true)
     {
             if (IsPrime(i))
             yield return i;
             i++;
     }
 }
```

Dado el método textttGetPrimes que devuelve una lista de números primos, nos fue pedido modificarlo para que de una forma "parara" su ejecución. Para esto implementamos el método iterador textttGetPrimes que devuelve un `IEnumerable<int>`, para su implementación utilizamos el comando `yield return`. Cuando la instrucción que contiene al `yield return` es alcanzada, una expresión computada es devuelta, en este caso un número primo, y es guardada la posición en la que se detuvo la ejecución al retornar. La ejecución es reiniciada desde esa posición la próxima vez que se llame al iterador. Luego no hay un ciclo infinito, hay una función/iterador que puede ser llamada una cantidad infinita de veces.

Pregunta 3.

¿Por qué la siguiente sentencia no bloquea el programa?

```
GetPrimes().Where(prime =>
                        prime.ToString().StartsWith("2")).Take(10);
```

Solución:
Al ser `GetPrimes` un método lazy este genera elementos a medida que son requeridos. El método `Where` pide elementos a `GetPrimes()` si estos pasan el predicado se ejecuta el `Take`. El método Where recibe un delegado a función, y filtra una secuencia de valores devolviendo un `IEnumerable<T>` que contiene los valores que satisfacen el predicado. Take recibe un entero `x` y devuelve un número de `x` elementos contiguos desde el inicio de una secuencia dada.

La sentencia anterior es un ejemplo de fluent programming, C# usa fluet programming en LINQ para construir queries usando "operadores estándar de queries", su implementación está basada en métodos extensores.

Pregunta 4.

Convierta el siguiente código Haskell a C#:

```
Four :: Integer -> Integer
Four x = 4
Infinity :: Integer
Infinity = 1 + Infinity
```

Solución:

Delegados, Delegados Anónimos y Expresiones Lambda.

Los delegados son tipos que representan referencias a métodos con una lista de parámetros particulares y un tipo de retorno específico. Al inicializar uno, se puede asociar su instancia con un m'etodo compatible en signatura y tipo de retorno. Dicho método puede invocarse a través de la instancia del delegado.

Los delegados resultan muy útiles para pasar métodos como argumento hacia otros métodos. Los handlers no son más que métodos que se invocan a través de delegados.

El código siguiente muestra un ejemplo de como declarar un delegado **??**

```
public delegate int PerformCalculation(int x, int y);
```

Cualquier método de alguna clase o struct accesible, cuyos parámetros y tipo de retorno coincidan con los del delegado puede ser asignado a este, da igual si es método de estático o de instancia.

La habilidad de poder referirnos a métodos como parámetros hace a los delegados ideales como callback methods(precisamente métodos que se pasa a otro con un cierto fin). Un ejemplo clásico es el de la referencia a un método que compare dos objetos, cuya utilidad sería pasarse como parámetro a un algoritmo de ordenación. Esto permite escribir el algoritmo de ordenación de frma más genérica.

```
public delegate int Compare<T>(int x, int y);
```

Para mayor facilidad C# ofrece una serie de delegados predefinidos con numerosas sobrecargas. El siguiente ejemplo(los Func en general) especifica una función que recibe una serie de parámetros Ti y retorna un TResutl, mientras que los Action reciben una serie de parámetor igual, pero realizan una secuencia de acciones sin retornar nigún valor.

```
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1,
    T2 arg2);
public delegate TResult Action<in T1, in T2, in T3>(T1 arg1, T2
    arg2, T3 arg3);
```

Existen también los llamados delegados anónimos, que son aquellos que no necesitan estar definidos como un método de ninguna clase. Un ejemplo sería al llamar a una función Ordena como la referida anteriormente, que necesitaba un delegado que supiera ordenar par de elementos del mismo tipo, en vez de llamar al delegado definido podríamos hacer los siguiente:

```
Ordenar(emailList, delegate(Email e1, Email e2){return e1.Subject.
    CompareTo(e2.Subject);});
```

Por si fuera poco, para hacernos la vida aún más fácil, existen las expresiones lambda. Estas no son más que expresiones de la forma siguiente:

1. Expresión lambda con una expresión como cuerpo

```
(input-parameters) => expression
```

2. Expresión lambda con un conjunto de sentencias como cuerpo

```
(input-parameters) => {sequence-of-statements}
```

Cualquier expresión lambda puede convertirse en delegado. El tipo correspondiente depende de los tipos de los parámetros y de los valores de retorno, osea, si no retorna ningún valor se lleva al tipo de delegado Action, en otro caso a Func. Por ejemplo, si la expresión lambda tiene dos parámetros y ningún valor de retorno se convertirá en Action<T1, T2>, en cambio una que posea un parámetro y retorne un valor pasaría a Func<T,TResult>.

Con expresiones lambdas el método de ordenación del que se hablaba quedaría:

```
Ordenar(emailList, (Email e1, Email e2) => e1.Subject.CompareTo(e2
    .Subject));
```

Debe destacarse que siempre que el compilador pueda inferir el tipo de los parámetros de la expresión lambda no es necesario agregarlos. En este caso como el método Ordena recide una lista de Emails, por lo que podemos ahorrarnos especificar el tipo de e1 y e2:

```
Ordenar(emailList, ( e1, e2) => e1.Subject.CompareTo(e2.Subject));
```

Haskell y los lenguajes funcionales. Lazy evaluation.

Haskell es un lenguaje de programación estandarizado multi-propósito puramente funcional con semánticas no estrictas y fuerte tipificación estática.

Lazy Evaluation o call-by-need como muchos la llaman es una característica de algunos lenguajes de programación que permite que un bloque de código se evalúe en forma tardía, o como bien lo dice lo dice su nombre, cuando se necesite. La evaluación perezosa proviene del paradigma funcional. Es una característica de los lenguajes funcionales como Haskell, así como de otros multiparadigmas como Scala.

Hechas estas observaciones podemos decir entonces que el código anterior se traduce en C# como:

```
Func<Func<int>, int> four = x => 4;
Func<int> infinity = null;
infinity = () => infinity() + 1;
```

Pregunta 5.

¿Cuál es el resultado de evaluar `Infinity` en `Four`?

Como ya se mencinó la evaluación en Haskell es lazy, lo que significa que no se realiza ninguna evaluación a menos que sea necesario. Por ello, como la función Four no realiza ninguna evaluación del parámetro de entrada, la función Infinity no será forzada a evaluarse y el resultado de Four será 4. En caso de evaluarse la función Infinity se lanzaría un mensaje de error, puesto que esta no recibe un parámetro de entrada.

Pregunta 6.

¿Son equivalentes los siguientes códigos?

Código 1:

```
if (Cond1() || Cond2())
{
    Console.WriteLine(true);
}
else
{
    Console.WriteLine(false);
}
```

Código 2:

```
if (Cond1() | Cond2())
{
    Console.WriteLine(true);
}
else
{
    Console.WriteLine(false);
}
```

Los códigos anteriores no son equivalentes ya que en el **Código 1** si `Cond1()` resulta verdadero `Cond2()` no se va evaluar y en el caso del **Código 2**, independientemente del resultado de la evaluación de `Cond1()` también se evaluará `Cond2()`.

Operador lógico | :

El operador | computa el OR lógico de los operandos. El resultado de realizar `x | y` es verdadero si cualquiera de los dos, `x` o `y`, evalúan verdadero. En otro caso el resultado es falso. El operador | evalúa ambos operandos incluso cuando el operando de la izquierda evalúa verdadero, por lo que la operación resulta verdadera independientemente de lo que evalúe el operando de la derecha.

En el siguiente ejemplo, el operando de la derecha de | es una llamada a un método, la cual es realizada independientemente del resultado del operador de la izquierda.

```
bool SecondOperand()
{
Console.WriteLine("Second operand is evaluated.");
return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
```

```
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

Operador lógico condicional || :

El operador lógico condicional || computa el OR lógico de sus operandos. El resultado de || es verdadero is `x` o `y` evalúa verdadero. En otro caso el resultado es falso. Si `x` evalúa verdadero, `y` no se evalúa.

En el siguiente ejemplo el operando de la derecha de `||` es una llamada a un método, la cual no es realizada si el operando de la izquierda evalúa verdadero.

```
bool SecondOperand()
{
Console.WriteLine("Second operand is evaluated.");
return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

Pregunta 7.

Explique cómo funciona `yield return`. ¿Cómo se logra este comportamiento? ¿En Java existe algún mecanismo análogo?

Los iteradores son como funciones, pero en lugar de retornar un solo valor devuelven una secuencia de valores, uno a uno. Si se emplea la versión no genérica de `IEnumerator` los valores devueltos serán de tipo `object`.

Para una correcta implementación de un iterador se necesita mantener estados internos para saber por donde se encuentra la iteración mientras se enumera la colección. Los iteradores son transformados por el compilador en una máquina de estados que guarda la posición actual y sabe como moverse “solo” para la próxima posición.

La declaración de `yield return` devuelve un valor cada vez que el iterador lo encuentra, enseguida se retorna a quien hizo el llamado el elemento solicitado. Cuando se pide el próximo elemento el código se comienza a ejecutar inmediatamente siguiendo al `yield return` previamente ejecutado.

Iteradores y estados:

Cuando `GetEnumerator()` es llamado por primera vez en un `foreach` se crea un objeto iterador y su estado es inicializado en un "start" especial que representa el hecho de que ningún código ha sido ejecutado en el iterador y por lo tanto ningún valor ha sido devuelto. El iterador mantiene su estado mientras el `foreach` en el llamado continúe ejecutándose. Cada vez que el ciclo pida el próximo valor se entra al iterador y se continúa donde se quedó la vez anterior por el ciclo; la información del estado almacenada en el objeto iterador es usada para determinar desde donde se debe continuar. Cuando `foreach` culmina en el llamado el estado del iterador se deja de guardar.

Siempre es seguro llamar a `GetEnumerator()` nuevamente, nuevos objetos `enumerators` serán creados cuando sea necesario.

Ejemplos de implementación de GetEnumerator() están disponibles en la carpeta Codes(CSharpBuiltInTypes.cs y Program.cs a partir de la línea 39).

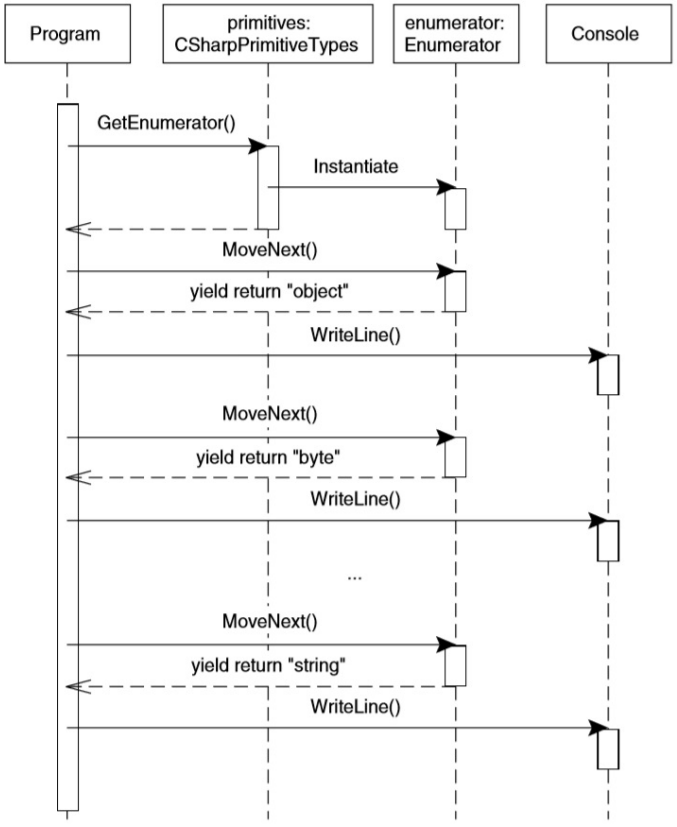

Figura 1: Diagrama de sequencias de `yield return`

En el ejemplo que se encuentra en la carpeta Codes(clase Program línea 39) el `foreach` inicia un llamado a `GetEnumerator()` en la instancia de `CSharpBuiltInTypes` llamada `keywords`. Dada la instancia iteradora (referenciada por `iterator`), `foreach` comienza cada iteración con un llamado a `MoveNext()`. En el iterador se devuelve un valor al `foreach` donde se llamó. Después del `yield return`, el método `GetEnumerator()` aparentemente se pausa hasta el próximo `MoveNext()`. De regreso al cuerpo del ciclo, el `foreach` muestra el valor devuelto en la pantalla. En el ciclo regresa y vuelve a llamar a `MoveNext()` en el iterador. La segunda vez se retoma en el segundo `yield return`. Nuevamente el `foreach` muestra lo que devolvió `CSharpBuiltInTypes` y comienza otra vez el ciclo. El proceso continúa hasta que no haya más `yield return` en el iterador. En ese punto el ciclo `foreach` en el llamado termina porque `MoveNext()` retorna `false`.

Observaciones:

- Cuando se usa `yield` en una declaración, se indica que el método, operador o parte `get` en la que aparece es un iterador. Usando `yield` para definir un iterador se elimina la necesidad de crear explícitamente una clase extra (la clase que contiene el estado para una enumeración) cuando se implementa `IEnumerable` y `IEnumerator` para una colección
- Se emplea `yield return` para retornar elemento por elemento
- La sequencia retornada por un método iterador puede ser obtenida usando `foreach` o `LINQ query`. Cada iteración del ciclo `foreach` llama al método iterador, la expresión es retornada y se guarda la posición actual en el código. La ejecución es reiniciada desde esa posición la próxima vez que sea llamada la función iteradora
- Se pude usar `yield break` para finalizar la iteración

Requerimientos que debe cumplir la declaración de un iterador:

- El tipo de retorno debe ser `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, `IEnumerator<T>`
- La declaración no puede tener parámetros `ref` o `out`

El tipo de un iterador que retorna `IEnumerable` o `IEnumerator` es `object`. Si el iterador retorna `IEnumerable<T>` o `IEnumerator<T>`, debe haber una conversión implicita del tipo de la expresión en la declaración del `yield return` al parámetro de tipo genérico.

No se puede incluir `yield return` o `yield break` en expresiones lambda o en métodos anónimos y en métodos que contienen "unsafe blocks"

Otros requerimientos de `yield return`:

- Si se usa un llamado a `yield return`, el compilador de C# genera el código necesario para mantener el estado del iterador
- Si se emplea `return` en vez de `yield return` el programador es responsable de mantener su propia máquina de estado y de retornar una instancia de una de las interfaces del iterador
- En un iterador todos los "code path" deben tener al llamado `yield return` si van a retornar algún dato

Manejo de excepciones:

- Una declaración de `yield return` no puede estar ubicado en un bloque `try-catch`, pero si en uno `try-finally`
- Un `yield break` pude estar en un bloque `try` o en uno `catch` pero no en uno `finally`
- Si el cuerpo del `foreach` (fuera del método iterador) lanza una excepción, un bloque `finally` en el método iterador es ejecutado

Implementación:

```
IEnumerable<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
        ...
}
```

El llamado a `MyIteratorMethod` no ejecuta el cuerpo del método, sino que el llamado retorna un `IEnumerable<string>` en la variable `elements`.

En una iteración del ciclo `foreach`, el método `MoveNext()` es llamado por `elements`. Este llamado ejecuta el cuerpo de `MyIteratorMethod` hasta que el próximo `yield return` se alcanzado. La expresión retornada por el `yield return` determina no solo el valor de la variable `element` para el consumo del cuerpo del ciclo sino la propiedad de `elements` `Current`, la cual es un `IEnumerable<string>`.

En cada iteración subsequente del `foreach` la ejecución del cuerpo del iterador continúa desde donde se quedó, nuevamente deteniéndose cuando alcanza el `yield return`. El ciclo `foreach` es completado cuando se llega al final del método iterador o a un `yield break`.

Ejemplo del uso de `yield return` en un ciclo `for`:

El siguiente ejemplo tiene una declaración de `yield return` dentro de un ciclo `for`. Cada iteración del cuerpo del `foreach` en el método `Main` crea una llamada a la función iteradora `Power`. Cada llamada a la función iteradora procede a la próxima ejecución del `yield return`, lo cual ocurre durante la próxima iteración del ciclo `for`.

El tipo de retorno del iterador es `IEnumerable`, el cual es un tipo de interfaz iteradora. Cuando el método iterador es llamado retorna un objeto enumerable que contiene las potencias de un número.

```
public class PowersOf2
{
        static void Main()
        {
                // Display powers of 2 up to the exponent of 8:
                foreach (int i in Power(2, 8))
                        Console.Write("{0} ", i);
        }

        public static IEnumerable<int> Power(int number, int
           exponent)
        {
                int result = 1;
                for (int i = 0; i < exponent; i++)
                {
                        result = result * number;
                        yield return result;
                }
        }
        // Output: 2 4 8 16 32 64 128 256
}
```

Este y otros ejemplos de iteradores se encuentran en la carpeta Codes(PowersOf2.cs y GalaxyClass.cs).