Seminario de Lenguajes de Programación
C# (Primera Parte)

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

16 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.

```
using System.Linq;
using System.Text ;
using System;
namespace CustomExtensions
{
        // 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[ ]  {' ',  '. ',
                            '?'},  StringSplitOptions.
                           RemoveEmptyEntries().Length;
                }
        }
}
```

```
namespace Extension_Methods_Simple
{
        // Import the extension method namespace.
        using CustomExtensions;
        class Program
        {
                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();
                System. 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>`. Ilustremos lo anterior con un ejemplo.

Ejemplo

```
public class Student
    {
        public string Name { get; set; }

        public int Group { get; set; }

        public Student(string name, int group)
        {
            Name = name;
            Group = group;
        }
    }

static void Main(string[] args)
        {
            List<Student> students = new List<Student>
            {
```

```
            new Student("Amalia", 312),
            new Student("Gabriela",312),
            new Student("Sandra",311),
            new Student("Paula",311)
        };
        //groups is of type IEnumerable<IGrouping<TKey,
            TElement>>
        var groups = students.GroupBy(e => e.Group);

        foreach (var group in groups)
        {
            Console.WriteLine($"Estudiantes del grupo {group.
                Key}:");
            foreach (var student in group)
            {
                Console.WriteLine(student.Name);
            }
        }
        //Output
        //Esudiantes del grupo 312:
        //Amalia
        //Gabriela
        //Esudiantes del grupo 311:
        //Sandra
        //Paula
    }
```

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

Implementación de GroupBy no Lazy

```
public static IEnumerable<IGrouping<TKey, TSource>> MyGroupBy<
    TSource, TKey>(
            this IEnumerable<TSource> source,
            Func<TSource, TKey> keySelector)
        {
            List<IGroupableClass<TKey, TSource>> result = new List
                <IGroupableClass<TKey, TSource>>();
            List<TKey> keys = new List<TKey>();

            foreach (TSource ts in source)
            {
                TKey key = keySelector(ts);

                if (!keys.Contains(key))
                {
                    keys.Add(key);
                    result.Add(new IGroupableClass<TKey, TSource>(
                        key, FindGroup(key, keySelector, source)));
                }

            }
            return result;
        }

        public static IEnumerable<TSource> FindGroup<TKey, TSource
            >(TKey key, Func<TSource, TKey> keySelector,
            IEnumerable<TSource> source)
        {
            List<TSource> result = new List<TSource>();
            foreach (var s in source)
            {
                if (keySelector(s).Equals(key))
                    result.Add(s);
            }
            return result;
        }
```

Implementación de GroupBy Lazy

```
public static IEnumerable<IGrouping<TKey, TSource>> MyGroupByLazy<
    TSource, TKey>(
            this IEnumerable<TSource> source,
            Func<TSource, TKey> keySelector)
        {
            List<IGroupableClass<TKey, TSource>> result = new List
                <IGroupableClass<TKey, TSource>>();
```

```
            List<TKey> keys = new List<TKey>();

            foreach (TSource ts in source)
            {
                TKey key = keySelector(ts);

                if (!keys.Contains(key))
                {
                    keys.Add(key);
                    yield return new IGroupableClass<TKey, TSource
                        >(key, FindGroup(key, keySelector, source))
                        );
                }

            }
        }
```

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 $O(N^2)$.

Implementación de Take(K)

```
public static IEnumerable<TSource> MyTake<TSource>(
           this IEnumerable<TSource> source, int count)
        {
            int i = 0;
            foreach (var item in source)
            {
                if (i == count)
                    yield break;
                yield return item;
                i++;
            }
        }
```

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.

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, o sea, 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));
```

Pregunta 4.

Convierta el siguiente código Haskell a C#:

```
four :: Integer -> Integer
four x = 4
infinity :: Integer
infinity = 1 + infinity
```

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.

Podemos interpretar el código de Haskell anterior como que `four` es es una "función" que recibe un `Integer` y retorna un `Integer`, pero retorna el valor 4 independientemente de la entrada. Por otro lado, `infinity` sería como otra "función" que devuelve un `Integer` pero no recibe entrada alguna, cuya evaluación depende del retorno de un "llamado recursivo" a sí mismo.

Hechas estas observaciones podemos decir entonces que el código anterior se traduciría 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 mencionó 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 depende de la entrada para evaluarse a si misma, no se realiza ninguna evaluación del parámetro, la función `infinity` no será forzada a evaluarse y el resultado de `four` será 4. En caso de forzar la evaluación de la función `infinity` se lanzaría un mensaje de error, puesto que resulta imposible obtener una evalución de la misma (sería como un stack overflow =).

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?

`yield return`

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.

El siguiente ejemplo devuele los numeros del 0 al 9 en una sequencia, Inicialmente declaramos el método que devuelve el `IEnumerator` no genérico.

Ejemplo:

```
using System;
using System.Collections;

class Test
{
        static IEnumerator GetCounter()
        {
                for (int count = 0; count < 10; count++)
                        yield return count;
        }
}
```

Ahora observemos en lo que el compilador lo convierte:

```
internal class Test
{
        // Note how this doesn't execute any of our original code
        private static IEnumerator GetCounter()
        {
                return new <GetCounter>d__0(0);
        }

        // Nested type automatically created by the compiler to
           implement the iterator
        [CompilerGenerated]
        private sealed class <GetCounter>d__0 : IEnumerator<object
           >, IEnumerator, IDisposable
        {
                // Fields: there'll always be a "state" and "
                   current", but the "count"
                // comes from the local variable in our iterator
                   block.
                private int <>1__state;
                private object <>2__current;
                public int <count>5__1;

                [DebuggerHidden]
```

```
public <GetCounter>d__0(int <>1__state)
{
        this.<>1__state = <>1__state;
}

// Almost all of the real work happens here
private bool MoveNext()
{
        switch (this.<>1__state)
        {
                case 0:
                        this.<>1__state = -1;
                        this.<count>5__1 = 0;
                        while (this.<count>5__1 <
                            10)
                        {
                                this.<>2__current
                                    = this.<count>5
                                    __1;
                                this.<>1__state =
                                    1;
                                return true;
                                Label_004B:
                                this.<>1__state =
                                    -1;
                                this.<count>5__1
                                    ++;
                        }
                        break;

                case 1:
                        goto Label_004B;
        }
        return false;
}

[DebuggerHidden]
void IEnumerator.Reset()
{
        throw new NotSupportedException();
}

void IDisposable.Dispose()
{
}

object IEnumerator<object>.Current
{
        [DebuggerHidden]
```

```
                        get
                        {
                                return this.<>2__current;
                        }
                }

                object IEnumerator.Current
                {
                        [DebuggerHidden]
                        get
                        {
                                return this.<>2__current;
                        }
                }
        }
}
```

Comenzaremos comentando el código:

- El llamado `GetCounter()` llamado al constructor de `<GetCounter>d_0` el cual es el tipo implementando el iterador. El constructor solo setea el estado del iterador. Ningún código de la clase original ha sido ejecutado aún.
- varios elemetos del código están decorados con los atributos `CompilerGenerated` y `DebuggerHidden`. La documentación de MSDN para `CompilerGenerated` menciona que le permite a SQL Server acceso extra. `DebuggerHidden` solo para los "debuggers" (para aquellos que elijan tomar nota del atributo) de pasar o interrumpir el código
- `<GetCounter>d_0` (de ahora en adelante conocido como "el tipo iterador") implementa tres interfaces: `IEnumerator<object>`, `IEnumerator` y `IDisposable`. La primera implica las otras dos pero se anota para aclarar.
- Es muy importate que `IDisposable` sea implementado (aparte de que sea necesario para la implementación de `IEnumerator<object>`). El `foreach` de C# llama a `Dispose` en cualquier iterador que implemente `IDisposable`; el llamdo es un bloque `finally`. El los casos raros en que se usa un iterador manualmente en vez de un ciclo `foreach`, se debe usar la declaración `using` para aseguarr que se llame a `Disposable`.
- El tipo iterador es anidado, lo cual significa que tiene acceso a todos los miembros privados de la clase. Es importante si el bloque iterador llama a otros métodos privados o accede a variables privadas.
- El tipo iterador tiene tres campos: dos privados y uno público. Las dos varibles privadas manienen el registro del valor a retornar por la propiedad `Current` y el estado en que está el iterador.
- El constructor solo setea el estado del iterador. En este caso solo está llamando al constructor el método `GetCounter()` el cual pasa 0 en el estado inicial.
- `MoveNext()` es básicamente una declaración de cambio. El valor siempre está cambiando en su estado en la máquina de estado, así sabe que código tiene que ejecutar luego.

- El método `Reset` de `IEnumerator` siempre lanza `System.NotSupportedException`. Esto no solo es una decisón de implemntación, está en las especificaciones de C#.
- El el primer ejemplo simple, el método `Dispose` no hace nada.
- Ambas implementaciones de `IEnumerator` y `IEnumerator<object>` de `Cuerrent` simplemente retornan el valor de la variable `<>2_current`. Las especificaciones de lenguaje C# explícitamente deja el comportamiento de la propiedad `Current` indefinida en los casos "raros" (tales como acceder a esta antes del primer llamado a `MoveNext()` o cuando `MoveNext()` retorna `false`).

Interfaces genéricas vs. no-genéricas

Hemos visto como declarar un método para que retorne `IEnumerator` que implementa `IEnumerator<object>` también. Ahora cambiemos ligeramente el código para que retorne explícitamente `IEnumerator<int>`:

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

class Test
{
        static IEnumerator<int> GetCounter()
        {
                for (int count = 0; count < 10; count++)
                {
                        yield return count;
                }
        }
}
```

El método `GetCounter()` también cambia el tipo retorno:

```
private static IEnumerator<int> GetCounter()
{
        return new <GetCounter>d__0(0);
}
```

El iterador ahora implementa `IEnumerator<int>` en vez de `IEnumerator<object>`. el tipo involucrado aquí es llamado "*tipo yield*". Cada `yield return` tiene que retornar lago que pueda ser convertido implícitamente a tipo yield. Esta es la nueva signatura de la clase:

```
private sealed class <GetCounter>d__0 : IEnumerator<int>,
    IEnumerator, IDisposable
```

Similar para la propiedad `Current`:

```
private int <>2__current;

int IEnumerator<int>.Current
{
```

```
        get
        {
                return this.<>2__current;
        }
}
```

A parte de estos cambios menores la clase se mantiene igual.

Retornando `IEnumerable`

Hay algunos cambios más significativos si cambiaos el código original para retornar `IEnumerable` o su equivalente genérico en vez de `IEnumerator`. vamos a cambiar el código para retornar `IEnumerable<int>` y quedarnos con las interfaces genéricas de ahora en adelante, como hemos visto hace muy poca diferencia.

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

class Test
{
        static IEnumerable<int> GetCounter()
        {
                for (int count = 0; count < 10; count++)
                {
                        yield return count;
                }
        }
}
```

Ahora veamos el código resultante:

```
internal class Test
{
        private static IEnumerable<int> GetCounter()
        {
                return new <GetCounter>d__0(-2);
        }

        private sealed class <GetCounter>d__0 : IEnumerable<int>,
           IEnumerable, IEnumerator<int>, IEnumerator, IDisposable
        {
                // Fields
                private int <>1__state;
                private int <>2__current;
                private int <>l__initialThreadId;
                public int <count>5__1;

                public <GetCounter>d__0(int <>1__state)
```

```
{
        this.<>1__state = <>1__state;
        this.<>l__initialThreadId = Thread.
            CurrentThread.ManagedThreadId;
}

private bool MoveNext()
{
        switch (this.<>1__state)
        {
                case 0:
                        this.<>1__state = -1;
                        this.<count>5__1 = 0;
                        while (this.<count>5__1 <
                            10)
                        {
                                this.<>2__current
                                    = this.<count>5
                                    __1;
                                this.<>1__state =
                                    1;
                                return true;
                        Label_0046:
                                this.<>1__state =
                                    -1;
                                this.<count>5__1
                                    ++;
                        }
                        break;

                case 1:
                        goto Label_0046;
        }
        return false;
}

IEnumerator<int> IEnumerable<int>.GetEnumerator()
{
        if ((Thread.CurrentThread.ManagedThreadId
            == this.<>l__initialThreadId) && (this
            .<>1__state == -2))
        {
                this.<>1__state = 0;
                return this;
        }
        return new Test.<GetCounter>d__0(0);
}

IEnumerator IEnumerable.GetEnumerator()
```

```
                {
                        return ((IEnumerable<Int32>) this).
                            GetEnumerator();
                }

                void IEnumerator.Reset()
                {
                        throw new NotSupportedException();
                }

                void IDisposable.Dispose()
                {
                }

                int IEnumerator<int>.Current
                {
                        get
                        {
                                return this.<>2__current;
                        }
                }

                object IEnumerator.Current
                {
                        get
                        {
                                return this.<>2__current;
                        }
                }
        }
}
```

El caso de uso más común es que una instancia de `IEnumerable<T>` es creada, entonces algo (como una declaración `foreach`) llama a `GetEnumerator()` desde el mismo hilo, itera por la información, y deshecha el `IEnumerator<T>` al final. Dado el predominio de este patrón tiene sentido que el compilador de C# elija un patrón que optimice en vista a este caso. cuando ocurre este comportamiento solo se crea un objeto incluso cuando lo estamos usando para implementar dos instancias diferentes. El estado de -2 es usado paar representar que "`GetEnumerator` *no ha sido llamado aún*", mientras 0 es usado para representar "*Estoy listo para comenzar a iterar, pero* `MoveNext()` *no ha sido llamado*".

Sin embargo, si se trata de llamar a `GetEnumerator()`, desde un hilo diferente o cuando no está en el estado -2, el código tiene que crar una nueva instancia para mantener le seguimiento de los diferentes estados. En el último caso básicamente se tienen dos contadores independientes, por lo que necesitan almacenadores de información independientes. `GetEnumerator()` trata con la inicialización del iterador, y luego lo retorna listo. El aspecto de la seguridad del hilo está para prevenir dos hilos separados de llamados independientes de `GetEnumerator()` al mismo tiempo, y ambos terminado con el mismo iterador.

Este es el patrón básico cuando se trata de implementar `IEnumerable<T>`: el compilador implementa todas las inetrfaces en las misma clase, y el codigo perezosamente ("lazily") crea iteradores extra cuando es necesario.

Escogiendo entre interfaces para retornar

Normalmente `IEnumerable<T>` es la interfaz más flexible para retornar. Si el boque iterador no cambia nada, y la clase no está implementado `IEnumerable<T>` (en cuyo caso se tiene que retornar `IEnumerator<T>` del método `GetEnumerator()`), es una buena opción. Esto permite al cliente usar `foreach`, iterar varias veces, usar *LINQ to Objects*.

Manejo de estados.

Estados que el tipo iterador tiene que conocer:

- Su "puntero de instrucción virtual"
- Variables locales
- Valores iniciales de parámetros y `this`
- El hilo creado
- El último valor devuelto

Keeping track of where we've got to

El primer estado en la máquina de estados es el que sabe cuánto código ha sido ejecutado de la funete original (en un diagrama de una máquina de estados es el estado en el que se está actualmente), en ejemplo se ha visto como `<>1_state`. La especificación ace referencia a los estados *before, running, suspend* y *after*.

El bloque iterador no solo corre desde el principio al final. Cuando el método es llamdo originalmente, el iterador solo es creado. Solo cuando `MoveNext()` es llamado (después de un llamado a `GetEnumertor()` si se está usando `IEnumerable`). En este punto, la ejecución comienza en el principio del método como es lo normal, y progresa hasta la priemra declaración `yield return` o `yield break`, o el final del método. Entonces un valor booleano es retornado para indicar si el bloque termino o no de iterar. Si o cuando `MoveNext()` sea llamado otra vez, el método continúa ejecutándose a partir de lo siguinete al `yield return`. (Si la llamada previa terminó por algún motivo, se ha terminado de iterar y no pasará nada).

Valores que puede tomar `<>1_state`:

- -2: (Solo `IEnumerable`) Antes del primer llamado a `GetEnumerator()` desde el hilo creado
- -1: "Running" - el iterador es el código que se está ejecutando; también es sado para "After" - el iterador ha terminado, sea porque alcanzó el final del método o porque se encontró con un `yield break`

- 0: "Before" - `MoveNext()` no ha sido llamado todavía
- Cualquier número positivo: indica desde donde continuar; se devuelve al menos un valor y posiblemente se devolverán más. Los estados positivos también son usados cuando hay código corriendo aún, pero dentro de un bloque `try` con un bloque `finally` correspondiente.

Variables locales

Las variables locales normales son muy simples en un bloque iterador. Se convierten en variables de instanciada el el tipo iterador, y son asignadas con valores significativos de la misma forma (y en el mismo punto) que serían inicializadas en el código normal. Claro que siendo variables de instancia no hay ninguna idea de ellas siendo asignadas definitivamente, pero las reglas de compilación normales impiden ver sus verdadero valor.

La naturaleza de las variables locales significa que crear na instancia iteradora no requiere información extra acerca de las variables, cualquier valor inicial será seteado en el curso del código. El valor inicial puede depender de variables no-locales.

Parámetros y `this`

Los métodos implementados con con bloques iteradores pueden tomar parámentros, y su son métodos de instancia puede usar también `this`. Cualquier referencia a una variable de instancia del tipo contendor del bloque iterador es efectiva solo usando `this` y luego navegando de esa referencia a la variable.

`finally`

Iteradores poseen un problema, en vez de ejecutar todo el método antes de sacar de la pila, la ejecución se pausa cada vez que un valor es devuelto. No hay manera de garantizar que quien llama utilice de nuevo al iterador de alguna forma. Si se requiere que algún código sea ejecutado en algún punto luego de ser devuelto un valor esto no se puede garantizar. El código en un bloque `finally` que va a ser ejecutado normalmente en casi cualquier circunstancia antes de dejar el método no es tan confiable.

Vale recordar que los boques `finally` no son escritos explícitamente en C#, son generados por el compilador como parte de las declaraciones `lock` y `using`). `lock` es peligroso paraticularmente en un bloque iterador, en cualquier momento en se tenga un `yield return` dentro de un bloque `lock` se tiene un problema esperando. El código va a mantener el bloqueo incluso cuando se devuelva el próximo valor, y quién sabe cuando sea llamado `MoveNext()` o
texttttDispose() por el cliente. De la misma forma cualquier bloque `try/finally` que sean usados para asuntos críticos tales como seguridad no deberían aparecer en bloques iteradores: el cliente puede prevenir deliberadamente ejecutar el bloque `finally` si no necesita más valores.

Sin embargo, la máquina de estado es construida para que bloques `finally` sean ejecutados cuando un iterador es usado apropiadamente. Esto es porque `IEnumerator<T>` implementa `IDisposable`, y el ciclo `foreach` de C# llama a `Dispose` en los iteradores (incluso los `IEnumerator` no-genéricos, si no implementan `IDisposable`). La implementación de `IDisposable`

en el iterador generado decide los bloques `finally` que son relevantes para la posición actual (basado en el estado como siempre) y ejecuta el código apropiado.

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.

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

public class CSharpBuiltInTypes: IEnumerable<string>
{
        public IEnumerator<string> GetEnumerator()
        {
                yield return "object";
                yield return "byte";
                yield return "uint";
                yield return "ulong";
                //...
                yield return "string";
        }

        // The IEnumerable.GetEnumerator method is also required
        // because IEnumerable<T> derives from IEnumerable.

        System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator()
        {
                // Invoke IEnumerator<string> GetEnumerator()
                   above.
                return GetEnumerator();
        }
}

public class Program
{
        static void Main()
        {
                var keywords = new CSharpBuiltInTypes();
                foreach (string keyword in keywords)
```

```
                {
                        Console.WriteLine(keyword);
                }
        }
}
```

El código anterior retorna la siguiente sequencia:

```
object
byte
uint
ulong
...
string
```

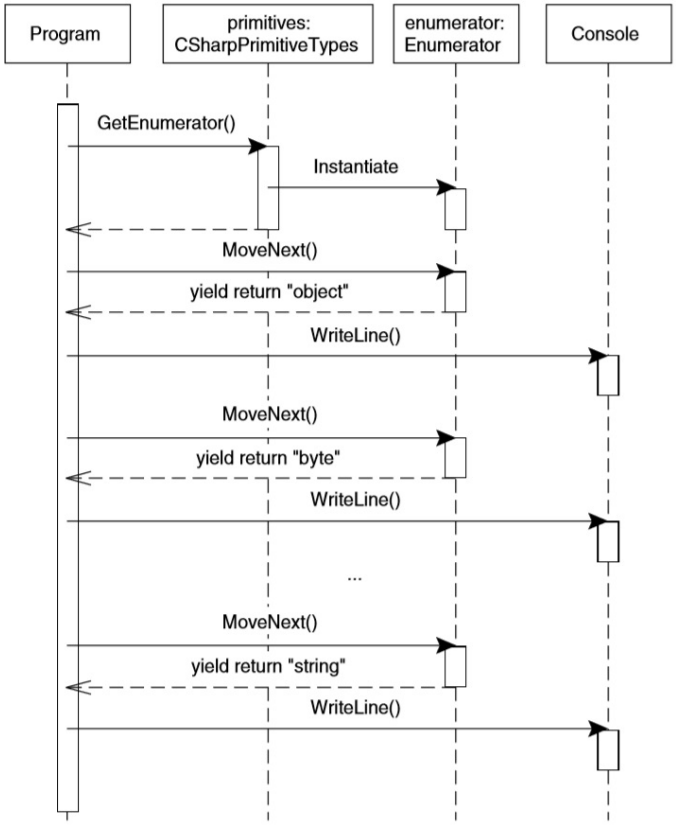

Figura 1: Diagrama de sequencias de `yield return`

En el ejemplo anterior 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 potescias 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 System.Collections.Generic.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
}
```

Mecanismo análogo al `yield return` en Java

En Java no existe el `yield return` pero se puede simular con el uso de los iteradores.

En Java existen tres iteradores y son usados para devolver elementos (uno por uno).

`Enumeration`:

Es una interfaz usada para obtener elementos de colecciones de herencia (Vector, Hashtable). `Enumeration` es el primer iterador presente de JDK 1.0 con mas funcionalidad. *Enumerations* son también usados para especificar la entrada a un *SequenceInputStream*. Se puede crear un objeto `Enumeration` llamando al método `elements()` de la clase vector de cualquier objeto vector.

```
// Here "v" is an Vector class object. e is of
// type Enumeration interface and refers to "v"
Enumeration e = v.elements();
```

Hay dos métodos en la interfaz `Enumeration`:

```
        // Tests if this enumeration contains more elements
        public boolean hasMoreElements();

        // Returns the next element of this enumeration
        // It throws NoSuchElementException
        // if no more element present
        public Object nextElement();
```

Ejemplo de `Enumeration`:

```
import java.util.Enumeration;
import java.util.Vector;

public class Test
{
        public static void main(String[] args)
        {
```

```
                // Create a vector and print its contents
                Vector v = new Vector();
                for (int i = 0; i < 10; i++)
                        v.addElement(i);
                System.out.println(v);

                // At beginning e(cursor) will point to
                // index just before the first element in v
                Enumeration e = v.elements();

                // Checking the next element availability
                while (e.hasMoreElements())
                {
                        // moving cursor to next element
                        int i = (Integer)e.nextElement();

                        System.out.print(i + " ");
                }
        }
}
```

Salida:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
```

Limitaciones de `Enumeration`:

- `Enumeration` es para clases herederas (vector, Hashtable). Por lo tanto no es un iterador universal
- No se pueden realizar opraciones de remover
- Solo se puede iterar hacia adelante

`Iterator`:

Es un iterador universal y se puede aplicar a cualquier objeto `Collection`. Usando `Iterator`, se puede leer y remover. Es una versión mejorada de `Enumeration` con funcionlidades adicinales de remover un elemento.

`Iterator` debe ser usado cuando se quiera enumerar elementos en todas las interfaces `Collection` implementadas como `Set, List, Queue, Deque` y también en todas las clases implementadas de `Map interface`. Es el único cursor disponible.

El objeto `Iterator` puede ser creado llamando al métdo `iterator()` pressente en la interfaz `Collection`.

```
// Here "c" is any Collection object. itr is of
// type Iterator interface and refers to "c"
Iterator itr = c.iterator();
```

La interfaz `Iterator` define tres métodos:

```
// Returns true if the iteration has more elements
public boolean hasNext();

// Returns the next element in the iteration
// It throws NoSuchElementException if no more
// element present
public Object next();

// Remove the next element in the iteration
// This method can be called only once per call
// to next()
public void remove();
```

El método `remove()` puede lanzar dos excepciones:

- `UnsupportedOperationException`: Si la operación remover no es soportada por este iterador
- `IllegalStateException`: Si el próximo método no ha sido llamado aún o el método `remove()` ya ha sido llamado después de la última llamada al próximo método

Ejemplo de `Iterator`:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Test
{
        public static void main(String[] args)
        {
                ArrayList al = new ArrayList();
                for (int i = 0; i < 10; i++)
                        al.add(i);

                System.out.println(al);
                // at beginning itr(cursor) will point to
                // index just before the first element in al
                Iterator itr = al.iterator();

                // checking the next element availabilty
                while (itr.hasNext())
                {
                        //  moving cursor to next element
                        int i = (Integer)itr.next();

                        // getting even elements one by one
                        System.out.print(i + " ");

                        // Removing odd elements
                        if (i % 2 != 0)
```

```
                    itr.remove();
                }
                System.out.println();
                System.out.println(al);

        }
}
```

Salida:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
[0, 2, 4, 6, 8]
```

Limitaciones de `Iterator`:

- Solo se puede iterar hacia delante
- No soporta reemplazo y adición de un nuevo elemento

`ListIterator`:

Solo es plicable para clases implamentadas de `List collection` como `arraylist, linkedlist`, etc. Provee iteración bidireccional.

`ListIterator` debe ser usado cuando se quiere enumerar elementos de `List`. este cursor tiene más funcionalidades (métodos) que `Iterator`.

El objeto `ListIterator` puede ser creado llamando al método `listIterator()` presente en la interfaz `List`.

```
// Here "l" is any List object, ltr is of type
// ListIterator interface and refers to "l"
ListIterator ltr = l.listIterator();
```

La interfaz `ListIterator` extiende la interfaz `Iterator`. Por lo que los tres métodos de `Iterator` están disponibles para `ListIterator`. Además hay seis métodos más.

```
// Forward direction

// Returns true if the iteration has more elements
public boolean hasNext();

// same as next() method of Iterator
public Object next();

// Returns the next element index
// or list size if the list iterator
// is at the end of the list
public int nextIndex();

// Backward direction

// Returns true if the iteration has more elements
```

```
// while traversing backward
public boolean hasPrevious();

// Returns the previous element in the iteration
// and can throws NoSuchElementException
// if no more element present
public Object previous();

// Returns the previous element index
//  or -1 if the list iterator is at the
// beginning of the list
public int previousIndex();

// Other Methods

// same as remove() method of Iterator
public void remove();

// Replaces the last element returned by
// next() or previous() with the specified element
public void set(Object obj);

// Inserts the specified element into the list at
// position before the element that would be returned
// by next(),
public void add(Object obj);
```

Claarmente los tres métodos que `ListIterator` hereda de `Iterator (hasNext(), next(), and remove())` hacen exactamente lo mismo en ambas interfaces. `hasPrevious()` y las opercio-nes anteriores son análogas a `hasNext()` y `next()`. Las operaciones *former* refieren al elemento antes del cursor. Las operaciones *previous* mueve el cursor hacia atrás, mientras que *next* lo mueve hacia delante.

`ListIterator` no tiene elemento *current*, su cursor siempre está entre el elemento que va a ser retornado por una llamada a `previous()` y el elemento que va a ser retornado or una llamada a `next()`.

El método `set()` puede lanzar cuatro excepciones:

- `UnsupportedOperationException`: Si la operacion `set` no es soportada por este `listIterator`
- `ClassCastException`: Si la clase del elemento especificado impide de ser añadida a la lista
- `IllegalArgumentException`: Si algún aspecto especificado impide ser añadido a la lista
- `IllegalStateException`: Si no han sido llamados `next` o `previous`, o `remove` o `add` han sido llamados después de la última llamada a `next` o `previous`

El método `add()` puede lanzar tres excepciones:

- `UnsupportedOperationException`: Si la operacion `add` no es soportada por este `listIterator`
- `ClassCastException`: Si la clase del elemento especificado impide de ser añadida a la lista

- `IllegalArgumentException`: Si algún aspecto especificado impide ser añadido a la lista

Ejemplo de `ListIterator`:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Test
{
        public static void main(String[] args)
        {
                ArrayList al = new ArrayList();
                for (int i = 0; i < 10; i++)
                        al.add(i);

                System.out.println(al);

                // at beginning ltr(cursor) will point to
                // index just before the first element in al
                ListIterator ltr = al.listIterator();

                // checking the next element availabilty
                while (ltr.hasNext())
                {
                        //  moving cursor to next element
                        int i = (Integer)ltr.next();

                        // getting even elements one by one
                        System.out.print(i + " ");

                        // Changing even numbers to odd and
                        // adding modified number again in
                        // iterator
                        if (i%2==0)
                        {
                                i++;  // Change to odd
                                ltr.set(i);  // set method to
                                   change value
                                ltr.add(i);  // to add
                        }
                }
                System.out.println();
                System.out.println(al);
        }
}
```

Salida:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
[1, 1, 1, 3, 3, 3, 5, 5, 5, 7, 7, 7, 9, 9, 9]
```

Limitaciones de `ListIterator`:

- Es el iterador más poderoso pero solo es aplicable para clases qque implementan `List`, por lo que no es un iterador universal

Programa de Java que demuestra la referencia de iteradores

```
import java.util.Enumeration;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Vector;

public class Test
{
        public static void main(String[] args)
        {
                Vector v = new Vector();

                // Create three iterators
                Enumeration e = v.elements();
                Iterator  itr = v.iterator();
                ListIterator ltr = v.listIterator();

                // Print class names of iterators
                System.out.println(e.getClass().getName());
                System.out.println(itr.getClass().getName());
                System.out.println(ltr.getClass().getName());
        }
}
```

Salida:

```
java.util.Vector$1
java.util.Vector$Itr
java.util.Vector$ListItr
```