

Investigating Iterators

Bart J.F. De Smet
bartde@outlook.com



pluralsight 
hardcore dev and IT training

Quick Recap of Iterators

■ Methods implementing enumerable sequences (C# 2.0)

- Return `IEnumerable<T>` or `IEnumerator<T>`
- Lazy evaluation triggered by iteration
 - “yield return” to produce the next element in the sequence
 - “yield break” to terminate the sequence

```
static IEnumerable<int> Numbers()  
{  
    var i = 0;  
    while (true)  
        yield return i++;  
}
```

Infinite loop evaluated lazily

■ Basis of LINQ to Objects (C# 3.0)

- Query operators over enumerable sequences
- Conveniently implemented as iterators

Implementing a Query Operator

- Where filter using a predicate function

- Implementation using iterator

```
static IEnumerable<T> Where<T>(this IEnumerable<T> source
                                Func<T, bool> predicate)
{
    foreach (T item in source)
        if (predicate(item))
            yield return item;
}
```

- Sample execution

```
var xs = new[] { 1, 2, 3, 4, 5, 6 };
var ys = xs.Where(x => x % 2 == 0);
foreach (var y in ys)
    Console.WriteLine(x);
```

Consumer drives evaluation

The State Machines Behind Iterators

- **Yield statements act as instruction pointers**
 - Consumer tries to move the pointer using MoveNext
 - E.g. iteration pattern of “foreach”
 - Producer runs until it hits a yield statement
 - “yield return” → MoveNext returns true and Current property is set
 - “yield break” → MoveNext returns false

```
static IEnumerator<int> Produce() {  
    yield return 2;  
    yield return 5;  
}
```

```
var xs = Produce();  
Debug.Assert( xs.MoveNext() && xs.Current == 2);  
Debug.Assert( xs.MoveNext() && xs.Current == 5);  
Debug.Assert(!xs.MoveNext());
```

The State Machines Behind Iterators

■ State machine generation

- Split iterator body into “basic blocks”
 - Ends with a yield statement
 - May be part of complex control flow
 - Each block is assigned a state

```
static IEnumerator<int> Produce() {  
    yield return 2;  
    yield return 5;  
}
```

```
class ProducerIterator : IEnumerator<int> {  
    private int _state;  
  
    public bool MoveNext() {  
        switch (_state) {  
            case 0: Current = 2; _state = 1; return true;  
            case 1: Current = 5; _state = 2; return true;  
            default: return false;  
        }  
    }  
  
    public int Current { get; private set; }  
}
```

yield return 2;

Object Lifetimes in Iterators

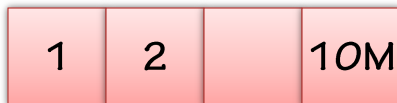
- Local variables are hoisted to the heap
 - Lifetime tied to iterator objects

```
static IEnumerable<int> Foo() {  
    byte[] bs;  
    yield return 42;  
    bs = new byte[10 * 1024 * 1024]; // 10MB  
    yield return 43;  
    yield return bs.Length;  
    yield return 44;  
}
```

Hoisted to heap

"bs" still alive here

```
class <Foo>__d0 {  
    int _state;  
    ...  
    byte[] bs;  
}
```



Tidbits About Iterators

■ Restrictions

- Iterators cannot have ref or out parameters
 - Reason: stack frame is hoisted to the heap
- Can't "yield return" in a try block of a try..catch statement
 - Rationale: may be confusing who handles *consumer* errors
 - One can "yield return" in a try block of a try..finally statement (or using)
- Iterators don't support Reset
 - Throws NotSupportedException

■ Concurrent usage

- Optimization of enumerable/enumerator pair
 - Single object for enumerable and enumerator
 - Lazy instantiation upon GetEnumerator call
- Thread checked using
 - Thread.ManagedThreadId
 - Environment.CurrentManagedThreadId (since .NET 4.5)

Tips and Tricks

- **Lazy evaluation for the win**
 - Eager – File.ReadAllLines() returns string[]
 - Lazy – File.ReadLines() returns IEnumerable<string>
- **Debugging LINQ to Objects queries**

```
static IEnumerable<T> Where<T>(this IEnumerable<T> source
                               Func<T, string> debug) {
    foreach (T item in source) {
        Console.WriteLine(debug(item));
        yield return item;
    }
}
```

Doesn't have to be bool!

Printf debugging

```
from x in xs
where "Before filter " + x
where f(x)
where "After filter " + x
select g(x)
```


Summary

- **Lazy evaluation for sequences**

- Avoids complexity of `IEnumerator<T>` implementation
- Used extensively in LINQ to Objects

- **State machine**

- Basic blocks based on “yield” points
- `MoveNext` by the consumer drives the state machine forward

- **Caveats**

- Laziness has a runtime cost
- Object lifetimes can be longer than expected