

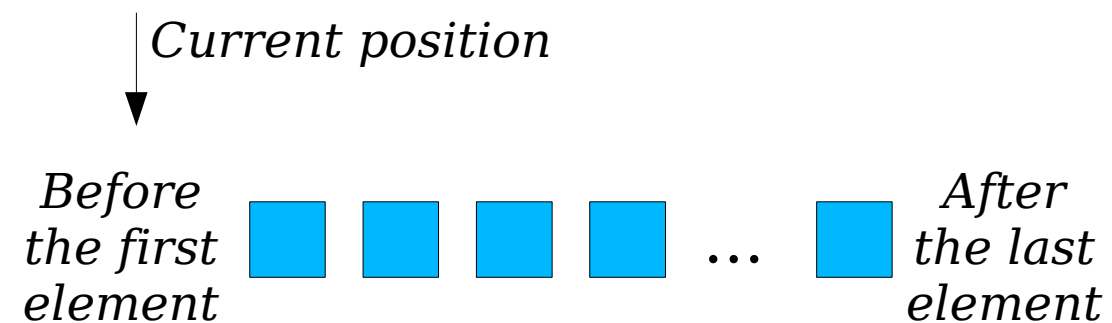
# Enumerables, Iterators and Yield

# .NET Enumerators

An enumerator works over a sequence, and is considered to have a concept of “before the first element”, the set of elements, and another concept of “after the last element”

An enumerator also has a current position, which is changed by calling the MoveNext method

An enumerator's Current property returns the element at the current position, or if accessed when the current position is “before the first element” or “after the last element”, throws an exception



# .NET Enumerators

`System.Collections.IEnumerable` has one method

- `IEnumerator GetEnumerator();`

`System.Collections.IEnumerator` has two methods and one property

- object `Current`: returns reference to the element at the current position (note the return type is `object` [not type safe])
- `void Reset()`: resets the current position to “before the first element”
- `bool MoveNext()`: either changes current to refer to the next element (and returns `true`), or “after the last element” if no more available (and returns `false`)

[Enumerator is initialised to “before the first element”, so `MoveNext` must be called to have `Current` refer to first element]

# .NET Enumerators

.NET also has type-safe generic versions of enumerable and enumerator

- `System.Collections.Generic.IEnumerable<T>` has one method
- `IEnumerator<T> GetEnumerator();`

`System.Collections.Generic.IEnumerator<T>` had one method and one property

- `bool MoveNext():` advances the current position
- `T Current:` returns type-safe reference to the element at the current position

Note there is no `Reset` method

# Notes on Enumerators

Enumerators are read-only

Used to read elements, but not to set them

Not synchronised

foreach keyword in C# and for each in VB requires that a collection supports the IEnumerable interface

An enumerator is only valid while the underlying sequence does not change

C# iterators work with the Enumerator concept

# IEnumerable vs. IEnumerator

Some people get confused over when to use which

- Think of instances of IEnumerable as factories for instances of IEnumerator (where the real work is done)
- The third object in the foreach statement must implement IEnumerable (not IEnumerator)
- Foreach (<type> <instance> in <IEnumerable>)

For collections, this can either be:

- the collection itself (assuming it implements IEnumerable) or
- a property within the collection of type IEnumerable or
- a method whose return value is IEnumerable

Reason for multiple enumerables on the same collection?

- Different selection algorithms (all, reverse list, every second item, every item that matches some search criteria, etc.)

# Contrast STL & C# Iterators

The term “iterator” is used in both C++'s well known & highly respected STL and in C#

In broad scope, they have a similar goal

To somehow enumerate though an underlying sequence of elements

.. but they work quite differently

- STL iterators can be considered as variables like pointers with interesting additional characteristics, and a sequence is defined by a pair of STL iterators
- C# iterators are statement blocks used to define methods
- There is iterator code inside a method declaration
- There is no such thing as a C# iterator variable
- Traversing the sequence is done inside the C# iterator code block

# Yield Statements

Implementations of enumerators are boilerplate, except for the MoveNext method (assuming we wish to be able to support a variety of selection criteria, need different MoveNext)

- MoveNext is called multiple times, each time moving the current position forward and returning the next available element, if available, or else an indicator that the end has been reached
- C# iterators are a way of implementing enumerators with minimum coding
- The compiler provides the boilerplate code, and you provide the MoveNext, in the form of yield statements
- yield return: returns next element
- yield break: end of sequence reached
- During compilation, enumerator type with appropriate MoveNext gets automatically created

# C# Iterators As Interface Impl

It is recommended that any collection type implements IEnumerable

```
class MyCollection<T> : IEnumerable<T>{
    static int initialMax = 10;
    int currentPos = 0;
    T[] innerArray = new T[initialMax];
    public void Add(T v){..}
    public IEnumerator<T> GetEnumerator(){
        for (int i = 0; i < currentPos; i++)
            yield return innerArray[i];
    }
    .. // to use
    foreach (int i in collection)
        Console.WriteLine(i);
```

# C# Iterators As Properties

Can also provide enumerable properties, for specific algorithms

```
public IEnumerable<T> GetEverySecondNum {
    get{
        for (int i = 0; i < currentPos; i +=2)
            yield return innerArray[i];
    }
    ...
    // to use
    foreach (int i in col.GetEverySecondNum)
        // foreach uses property that returns IEnumerable
        Console.WriteLine(i);
}
```

# C# Iterators As Methods

Can use iterators to create methods

```
public IEnumerable<T> FirstThreeItemsTwice () {  
    int max = 3;  
    if (max > currentPos) max = currentPos;  
    for (int i = 0; i < max; i++)  
        yield return innerArray[i];  
    // may have multiple loops within same iterator  
    for (int ix = 0; ix < max; ix++)  
        yield return innerArray[ix];  
}  
  
...  
foreach (MyIntData i in col.FirstThreeItemsTwice ())  
    Console.WriteLine(i.Num);
```

# How Iterators Work

How iterators work is internal to compiler

- Three states for active code: before (initial setting), running & after
- One state when not active: suspended

Clients should call MoveNext only when the object is in the before or suspended states

When MoveNext is called for an object in the before state:

- State changed to running
- Variables are initialized and any arguments read

When MoveNext is called for a suspended object:

- State changed to running
- Variables restored from time of previous suspend

# Iterator code executes

The Iterator code executes from the point of an earlier suspension, or from the beginning if this is the first time

Code stops running when one of these occurs:

- Yield return: iterator is suspended, current item is the yielded instance and MoveNext returns true
- Yield break: iterator state becomes final and MoveNext returns false; if within try blocks, the finally block(s) is/are run
- End of iterator block: iterator state becomes final and MoveNext returns false
- Exception: finally blocks, if any, are run, state changes to after and exception propagates to calling code

# Yield Break

Yield break is used to break out of loops and complete the iteration

```
public IEnumerable<T> MyModulus () {  
    for (int i = 0; i < currentPos; i++) {  
        if (i > 0 && i % 3 == 0) // condition for break  
            yield break;  
        else  
            yield return innerArray[i];  
    }  
}
```

# Using Methods within Iterators

If iterator code needs to invoke methods on elements, then like any generic code, the element must support that interface

```
interface IDoubleEngine<T>{
    T DoubleTheData ();
}

class MyDoublerCollection<T> where T : IDoubleEngine<T>{
    ...
    public IEnumerable<T> ListDoubles{
        get {for (int i = 0; i < currentPos; i++)
            yield return innerArray[i].DoubleTheData ();
        }
    }
}
```

# Manually Calling MoveNext

If you wish, client can manually call MoveNext

Usually will use foreach

```
IEnumerable<MyIntData> e = col.FirstThreeItems();  
IEnumerator<MyIntData> en = e.GetEnumerator();  
while (en.MoveNext())  
    Console.WriteLine(en.Current.Num);
```

# Summary of Yield

This is a C# construct, but plenty of developers are not aware of it

- Concept of coroutine
- Key to LINQ's deferred operators
- When working with an enumerator, often does need all elements in memory at once (smaller memory footprint), or often simply not interested in all elements (e.g. find first that matches a condition)
- When C# compiler detects yield keyword, it inserts a simple statement block to implement the coroutine

The Async C# async/await feature works in a similar way to yield!!