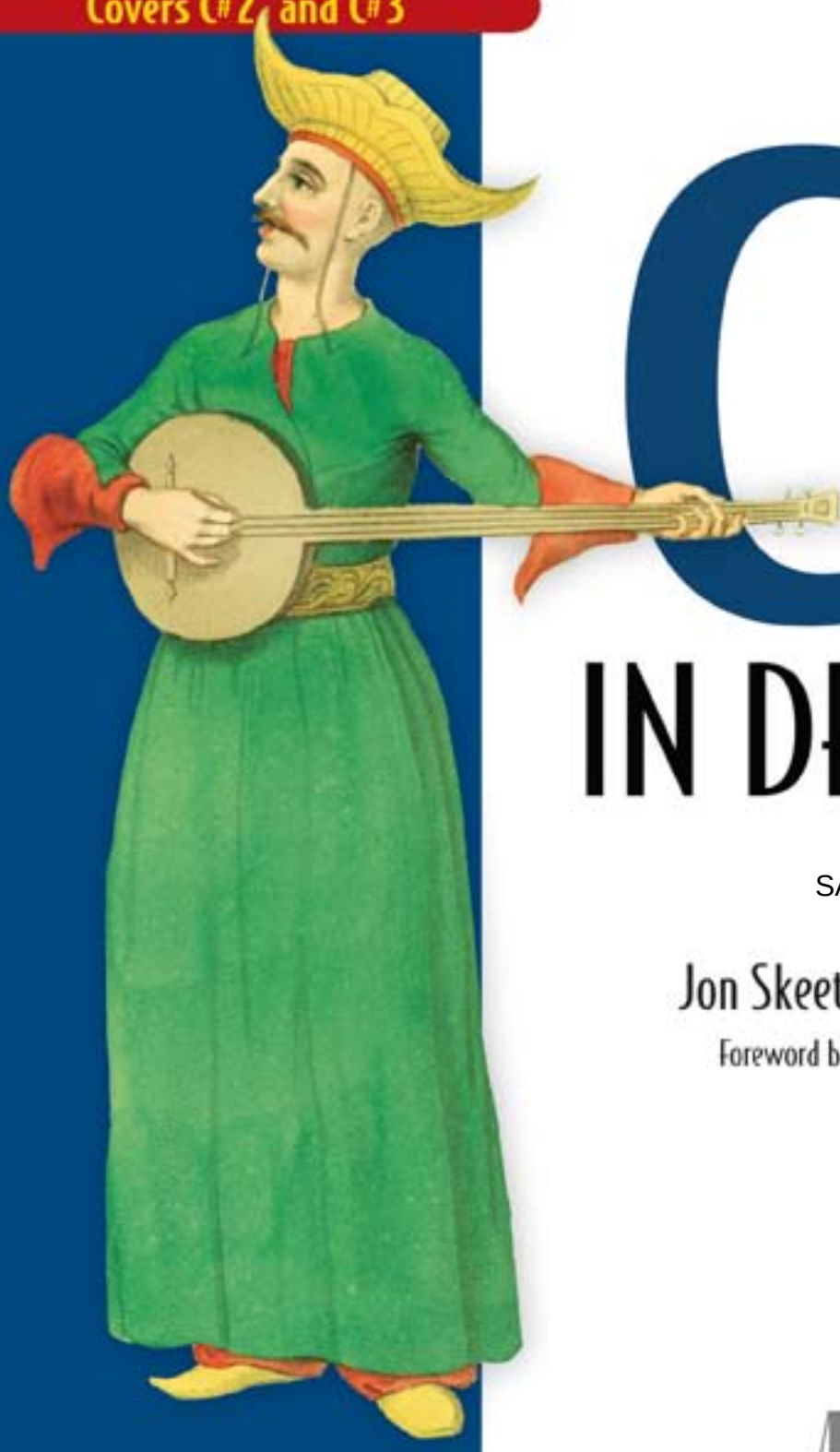
Covers C# 2 and C# 3



C# IN DEPTH

SAMPLE CHAPTER

Jon Skeet

Foreword by Eric Lippert

MANNING

C# in Depth
by Jon Skeet

Chapter 6

brief contents

6 Implementing iterators the easy way

This chapter covers

- Implementing iterators in C#1
- Iterator blocks in C#2
- A simple `Range` type
- Iterators as coroutines

The iterator pattern is an example of a *behavioral pattern*—a design pattern that simplifies communication between objects. It's one of the simplest patterns to understand, and incredibly easy to use. In essence, it allows you to access all the elements in a sequence of items without caring about what kind of sequence it is—an array, a list, a linked list, or none of the above. This can be very effective for building a *data pipeline*, where an item of data enters the pipeline and goes through a number of different transformations or filters before coming out at the other end. Indeed, this is one of the core patterns of LINQ, as we'll see in part 3.

EXTRA code available

In .NET, the iterator pattern is encapsulated by the `IEnumerator` and `IEnumerable` interfaces and their generic equivalents. (The naming is unfortunate—the pattern is normally called iteration rather than enumeration to avoid getting confused with

other meanings of the word *enumeration*. I've used *iterator* and *iterable* throughout this chapter.) If a type implements `IEnumerable`, that means it can be iterated over; calling the `GetEnumerator` method will return the `IEnumerator` implementation, which is the iterator itself.

As a language, C# 1 has built-in support for consuming iterators using the `foreach` statement. This makes it incredibly easy to iterate over collections—easier than using a straight `for` loop—and is nicely expressive. The `foreach` statement compiles down to calls to the `GetEnumerator` and `MoveNext` methods and the `Current` property, with support for disposing the iterator afterwards if `IDisposable` has been implemented. It's a small but useful piece of syntactic sugar.

In C#1, however, *implementing* an iterator is a relatively difficult task. The syntactic sugar provided by C#2 makes this much simpler, which can sometimes lead to the iterator pattern being worth implementing in cases where otherwise it would have caused more work than it saved.

In this chapter we'll look at just what is required to implement an iterator and the support given by C#2. As a complete example we'll create a useful `Range` class that can be used in numerous situations, and then we'll explore an exciting (if slightly off-the-wall) use of the iteration syntax in a new concurrency library from Microsoft.

As in other chapters, let's start off by looking at why this new feature was introduced. We'll implement an iterator the hard way.

6.1 C#1: The pain of handwritten iterators

We've already seen one example of an iterator implementation in section 3.4.3 when we looked at what happens when you iterate over a generic collection. In some ways that was harder than a real C#1 iterator implementation would have been, because we implemented the generic interfaces as well—but in some ways it was easier because it wasn't actually iterating over anything useful.

To put the C#2 features into context, we'll first implement an iterator that is about as simple as it can be while still providing real, useful values. Suppose we had a new type of collection—which can happen, even though .NET provides most of the collections you'll want to use in normal applications. We'll implement `IEnumerable` so that users of our new class can easily iterate over all the values in the collection. We'll ignore the guts of the collection here and just concentrate on the iteration side. Our collection will store its values in an array (`object[]`—no generics here!), and the collection will have the interesting feature that you can set its logical "starting point"—so if the array had five elements, you could set the start point to 2, and expect elements 2, 3, 4, 0, and then 1 to be returned. (This constraint prevents us from implementing `GetEnumerator` by simply calling the same method on the array itself. That would defeat the purpose of the exercise.)

To make the class easy to demonstrate, we'll provide both the values and the starting point in the constructor. So, we should be able to write code such as listing 6.1 in order to iterate over the collection.

Listing 6.1 Code using the (as yet unimplemented) new collection type

```
object[] values = {"a", "b", "c", "d", "e"};
IterationSample collection = new IterationSample(values, 3);
foreach (object x in collection)
{
   Console.WriteLine (x);
}
```

Running listing 6.1 should (eventually) produce output of "d", "e", "a", "b", and finally "c" because we specified a starting point of 3. Now that we know what we need to achieve, let's look at the skeleton of the class as shown in listing 6.2.

Listing 6.2 Skeleton of the new collection type, with no iterator implementation

```
using System;
using System.Collections;

public class IterationSample : IEnumerable
{
   object[] values;
   int startingPoint;

   public IterationSample (object[] values, int startingPoint)
   {
      this.values = values;
      this.startingPoint = startingPoint;
   }

   public IEnumerator GetEnumerator()
   {
      throw new NotImplementedException();
   }
}
```

As you can see, we haven't implemented `GetEnumerator` yet, but the rest of the code is ready to go. So, how do we go about implementing `GetEnumerator`? The first thing to understand is that we need to store some *state* somewhere. One important aspect of the iterator pattern is that we don't return all of the data in one go—the client just asks for one element at a time. That means we need to keep track of how far we've already gone through our array.

So, where should this state live? Suppose we tried to put it in the `IterationSample` class itself, making that implement `IEnumerator` as well as `IEnumerable`. At first sight, this looks like a good plan—after all, the *data* is in the right place, including the starting point. Our `GetEnumerator` method could just return `this`. However, there's a big problem with this approach—if `GetEnumerator` is called several times, several independent iterators should be returned. For instance, we should be able to use two `foreach` statements, one inside another, to get all possible pairs of values. That suggests we need to create a new object each time `GetEnumerator` is called. We *could* still implement the functionality directly within `IterationSample`, but then we'd have a class that didn't have a clear single responsibility—it would be pretty confusing.

Instead, let's create another class to implement the iterator itself. We'll use the fact that in C# a nested type has access to its enclosing type's private members, which means we can just store a reference to the "parent" `IterationSample`, along with the state of how far we've gone so far. This is shown in listing 6.3.

Listing 6.3 Nested class implementing the collection's iterator

```
class IterationSampleIterator : IEnumerator
{
   IterationSample parent;
   int position;

   internal IterationSampleIterator(IterationSample parent)
   {
      this.parent = parent;
      position = -1;
   }

   public bool MoveNext()
   {
      if (position != parent.values.Length)
      {
         position++;
      }
      return position < parent.values.Length;
   }

   public object Current
   {
      get
      {
         if (position==-1 ||
             position==parent.values.Length)
         {
            throw new InvalidOperationException();
         }
         int index = (position+parent.startingPoint);
         index = index % parent.values.Length;
         return parent.values[index];
      }
   }

   public void Reset()
   {
      position = -1;
   }
}
```

1 Refers to collection we're iterating over

2 Indicates how far we've iterated

3 Starts before first element

4 Increments position if we're still going

5 Prevents access before first or after last element

6 Implements wraparound

7 Moves back to before first element

What a lot of code to perform such a simple task! We remember the original collection of values we're iterating over ❶ and keep track of where we would be in a simple zero-based array ❷. To return an element we offset that index by the starting point ❻. In keeping with the interface, we consider our iterator to start logically before the first element ❸, so the client will have to call `MoveNext` before using the `Current` property for the first time. The conditional increment at ❹ makes the test

at ❺ simple and correct even if `MoveNext` is called again after it's first reported that there's no more data available. To reset the iterator, we set our logical position back to "before the first element" ❼.

Most of the logic involved is fairly straightforward, although there's lots of room for off-by-one errors; indeed, my first implementation failed its unit tests for precisely that reason. The good news is that it works, and that we only need to implement `IEnumerable` in `IterationSample` to complete the example:

```
public IEnumerator GetEnumerator()
{
   return new IterationSampleIterator(this);
}
```

I won't reproduce the combined code here, but it's available on the book's website, including listing 6.1, which now has the expected output.

It's worth bearing in mind that this is a relatively simple example—there's not a lot of state to keep track of, and no attempt to check whether the collection has changed between iterations. With this large burden involved to implement a simple iterator, we shouldn't be surprised at the rarity of implementing the pattern in C#1. Developers have generally been happy to use `foreach` on the collections provided by the framework, but they use more direct (and collection-specific) access when it comes to their own collections.

So, 40 lines of code to implement the iterator in C#1, not including comments. Let's see if C#2 can do any better.

6.2 C#2: Simple iterators with yield statements

I've always been the kind of person who likes to stay up until midnight on Christmas Eve in order to open a present as soon as Christmas Day arrives. In the same way, I think I'd find it almost impossible to wait any significant amount of time before showing you how neat the solution is in C#2.

6.2.1 Introducing iterator blocks and yield return

This chapter wouldn't exist if C#2 didn't have a powerful feature that cut down the amount of code you had to write to implement iterators. In some other topics the amount of code has only been reduced slightly, or has just made something more elegant. In this case, however, the amount of code required is reduced *massively*. Listing 6.4 shows the complete implementation of the `GetEnumerator` method in C#2.

Listing 6.4 Iterating through the sample collection with C#2 and `yield return`


```
public IEnumerator GetEnumerator()
{
   for (int index=0; index < values.Length; index++)
   {
      yield return values[(index+startingPoint)%values.Length];
   }
}
```

Four lines of implementation, two of which are just braces. Just to make it clear, that replaces the whole of the `IterationSampleIterator` class. Completely. At least in the source code… Later on we'll see what the compiler has done behind our back, and some of the quirks of the implementation it's provided, but for the moment let's look at the source code we've used.

The method looks like a perfectly normal one until you see the use of `yield return`. That's what tells the C# compiler that this isn't a normal method but one implemented with an *iterator block*. The method is declared to return an `IEnumerator`, and you can only use iterator blocks to implement methods[1] that have a return type of `IEnumerable`, `IEnumerator`, or one of the generic equivalents. The *yield type* of the iterator block is `object` if the declared return type of the method is a nongeneric interface, or the type argument of the generic interface otherwise. For instance, a method declared to return `IEnumerable<string>` would have a yield type of `string`.

No normal `return` statements are allowed within iterator blocks—only `yield return`. All `yield return` statements in the block have to try to return a value compatible with the yield type of the block. To use our previous example, you couldn't write `yield return 1;` in a method declared to return `IEnumerable<string>`.

NOTE *Restrictions on yield return*—There are a few further restrictions on yield statements. You can't use `yield return` inside a `try` block if it has any `catch` blocks, and you can't use either `yield return` or `yield break` (which we'll come to shortly) in a `finally` block. That doesn't mean you can't use `try/catch` or `try/finally` blocks inside iterators—it just restricts what you can do in them.

The big idea that you need to get your head around when it comes to iterator blocks is that although you've written a method that looks like it executes sequentially, what you've actually asked the compiler to do is create a *state machine* for you. This is necessary for exactly the same reason we had to put so much effort into implementing the iterator in C#1—the caller only wants to see one element at a time, so we need to keep track of what we were doing when we last returned a value.

In iterator blocks, the compiler creates a state machine (in the form of a nested type), which remembers exactly where we were within the block and what values the local variables (including parameters) had at that point. The compiler analyzes the iterator block and creates a class that is similar to the longhand implementation we wrote earlier, keeping all the necessary state as instance variables. Let's think about what this state machine has to do in order to implement the iterator:

- It has to have some initial state.
- Whenever `MoveNext` is called, it has to execute code from the `GetEnumerator` method until we're ready to provide the next value (in other words, until we hit a `yield return` statement).

[1] Or properties, as we'll see later on. You can't use an iterator block in an anonymous method, though.

- When the `Current` property is used, it has to return the last value we yielded.
- It has to know when we've finished yielding values so that `MoveNext` can return `false`.

The second point in this list is the tricky one, because it always needs to "restart" the code from the point it had previously reached. Keeping track of the local variables (as they appear in the method) isn't too hard—they're just represented by instance variables in the state machine. The restarting aspect is trickier, but the good news is that unless you're writing a C# compiler yourself, you needn't care about how it's achieved: the result from a black box point of view is that it just works. You can write perfectly normal code within the iterator block and the compiler is responsible for making sure that the flow of execution is exactly as it would be in any other method; the difference is that a `yield return` statement appears to only "temporarily" exit the method—you could think of it as being paused, effectively.

Next we'll examine the flow of execution in more detail, and in a more visual way.

6.2.2 Visualizing an iterator's workflow

It may help to think about how iterators execute in terms of a sequence diagram.[2] Rather than drawing the diagram out by hand, let's write a program to print it out (listing 6.5). The iterator itself just provides a sequence of numbers (0, 1, 2, –1) and then finishes. The interesting part isn't the numbers provided so much as the *flow* of the code.

Listing 6.5 Showing the sequence of calls between an iterator and its caller

```
static readonly string Padding = new string(' ', 30);

static IEnumerable<int> GetEnumerable()
{
   Console.WriteLine ("{0}Start of GetEnumerator()", Padding);

   for (int i=0; i < 3; i++)
   {
      Console.WriteLine ("{0}About to yield {1}", Padding, i);
      yield return i;
      Console.WriteLine ("{0}After yield", Padding);
   }

   Console.WriteLine ("{0}Yielding final value", Padding);
   yield return -1;

   Console.WriteLine ("{0}End of GetEnumerator()", Padding);
}

...

IEnumerable<int> iterable = GetEnumerable();
IEnumerator<int> iterator = iterable.GetEnumerator();
```

[2] See http://en.wikipedia.org/wiki/Sequence_diagram if this is unfamiliar to you.

```
Console.WriteLine ("Starting to iterate");
while (true)
{
   Console.WriteLine ("Calling MoveNext()...");
   bool result = iterator.MoveNext();
   Console.WriteLine ("... MoveNext result={0}", result);
   if (!result)
   {
      break;
   }
   Console.WriteLine ("Fetching Current...");
   Console.WriteLine ("... Current result={0}", iterator.Current);
}
```

Listing 6.5 certainly isn't pretty, particularly around the iteration side of things. In the normal course of events we'd just use a `foreach` loop, but to show exactly what's happening when, I had to break the use of the iterator out into little pieces. This code broadly does what `foreach` does, although `foreach` also calls `Dispose` at the end, which is important for iterator blocks, as we'll see shortly. As you can see, there's no difference in the syntax within the method even though this time we're returning `IEnumerable<int>` instead of `IEnumerator<int>`. Here's the output from listing 6.5:

```
Starting to iterate
Calling MoveNext()...
                      Start of GetEnumerator()
                      About to yield 0
... MoveNext result=True
Fetching Current...
... Current result=0
Calling MoveNext()...
                      After yield
                      About to yield 1
... MoveNext result=True
Fetching Current...
... Current result=1
Calling MoveNext()...
                      After yield
                      About to yield 2
... MoveNext result=True
Fetching Current...
... Current result=2
Calling MoveNext()...
                      After yield
                      Yielding final value
... MoveNext result=True
Fetching Current...
... Current result=-1
Calling MoveNext()...
                      End of GetEnumerator()
... MoveNext result=False
```

There are various important things to note from this output:

- None of the code we wrote in `GetEnumerator` is called until the first call to `MoveNext`.
- Calling `MoveNext` is the place all the work gets done; fetching `Current` doesn't run any of our code.
- The code stops executing at `yield return` and picks up again just afterwards at the next call to `MoveNext`.
- We can have multiple `yield return` statements in different places in the method.
- The code doesn't end at the last `yield return`—instead, the call to `MoveNext` that causes us to reach the end of the method is the one that returns `false`.

There are two things we haven't seen yet—an alternative way of halting the iteration, and how `finally` blocks work in this somewhat odd form of execution. Let's take a look at them now.

6.2.3 Advanced iterator execution flow

In normal methods, the `return` statement has two effects: First, it supplies the value the caller sees as the return value. Second, it terminates the execution of the method, executing any appropriate `finally` blocks on the way out. We've seen that the `yield return` statement temporarily exits the method, but only until `MoveNext` is called again, and we haven't examined the behavior of `finally` blocks at all yet. How can we *really* stop the method, and what happens to all of those `finally` blocks? We'll start with a fairly simple construct—the `yield break` statement.

ENDING AN ITERATOR WITH YIELD BREAK

You can always find a way to make a method have a single exit point, and many people work very hard to achieve this.[3] The same techniques can be applied in iterator blocks. However, should you wish to have an "early out," the `yield break` statement is your friend. This effectively terminates the iterator, making the current call to `MoveNext` return `false`.

Listing 6.6 demonstrates this by counting up to 100 but stopping early if it runs out of time. This also demonstrates the use of a method parameter in an iterator block,[4] and proves that the name of the method is irrelevant.

Listing 6.6 Demonstration of `yield break`

```
static IEnumerable<int> CountWithTimeLimit(DateTime limit)
{
   for (int i=1; i <= 100; i++)
   {
      if (DateTime.Now >= limit)
      {
         yield break;
      }
```

Stops if our time is up

[3] I personally find that the hoops you have to jump through to achieve this often make the code much harder to read than just having multiple return points, especially as `try/finally` is available for cleanup and you need to account for the possibility of exceptions occurring anyway. However, the point is that it can all be done.

[4] Note that methods taking `ref` or `out` parameters can't be implemented with iterator blocks.

```
        }
        yield return i;
    }
}
...

DateTime stop = DateTime.Now.AddSeconds(2);
foreach (int i in CountWithTimeLimit(stop))
{
    Console.WriteLine ("Received {0}", i);
    Thread.Sleep(300);
}
```

Typically when you run listing 6.6 you'll see about seven lines of output. The `foreach` loop terminates perfectly normally—as far as it's concerned, the iterator has just run out of elements to iterate over. The `yield break` statement behaves very much like a `return` statement in a normal method.

So far, so simple. There's one last aspect execution flow to explore: how and when `finally` blocks are executed.

EXECUTION OF FINALLY BLOCKS

We're used to `finally` blocks executing whenever we leave the relevant scope. Iterator blocks don't behave quite like normal methods, though—as we've seen, a `yield return` statement effectively pauses the method rather than exiting it. Following that logic, we wouldn't expect any `finally` blocks to be executed at that point—and indeed they aren't.

However, appropriate `finally` blocks *are* executed when a `yield break` statement is hit, just as you'd expect them to be when returning from a normal method.[5] Listing 6.7 shows this in action—it's the same code as listing 6.6, but with a `finally` block. The changes are shown in bold.

Listing 6.7 Demonstration of `yield break` working with `try/finally`

```
static IEnumerable<int> CountWithTimeLimit(DateTime limit)
{
    try
    {
        for (int i=1; i <= 100; i++)
        {
            if (DateTime.Now >= limit)
            {
                yield break;
            }
            yield return i;
        }
    }
    finally
    {
```

[5] They're also called when execution leaves the relevant scope without reaching either a `yield return` or a `yield break` statement. I'm only focusing on the behavior of the two `yield` statements here because that's where the flow of execution is new and different.

```
        Console.WriteLine ("Stopping!");
    }
}
...
DateTime stop = DateTime.Now.AddSeconds(2);
foreach (int i in CountWithTimeLimit(stop))
{
    Console.WriteLine ("Received {0}", i);
    Thread.Sleep(300);
}
```

Executes however the loop ends

The `finally` block in listing 6.7 is executed whether the iterator block just finishes by counting to 100, or whether it has to stop due to the time limit being reached. (It would also execute if the code threw an exception.) However, there are other ways we might try to avoid the `finally` block from being called… let's try to be sneaky.

We've seen that code in the iterator block is only executed when `MoveNext` is called. So what happens if we never call `MoveNext`? Or if we call it a few times and then stop? Let's consider changing the "calling" part of listing 6.7 to this:

```
DateTime stop = DateTime.Now.AddSeconds(2);
foreach (int i in CountWithTimeLimit(stop))
{
    Console.WriteLine ("Received {0}", i);
    if (i > 3)
    {
        Console.WriteLine("Returning");
        return;
    }
    Thread.Sleep(300);
}
```

Here we're not stopping early in the iterator code—we're stopping early in the code *using* the iterator. The output is perhaps surprising:

```
Received 1
Received 2
Received 3
Received 4
Returning
Stopping!
```

Here, code is being executed after the `return` statement in the `foreach` loop. That doesn't normally happen unless there's a `finally` block involved—and in this case there are two! We already know about the `finally` block in the iterator method, but the question is what's causing it to be executed. I gave a hint to this earlier on—`foreach` calls `Dispose` on the `IEnumerator` it's provided with, in its own `finally` block (just like the `using` statement). When you call `Dispose` on an iterator created with an iterator block before it's finished iterating, the state machine executes any `finally` blocks that are in the scope of where the code is currently "paused."

We can prove very easily that it's the call to `Dispose` that triggers this by using the iterator manually:

```
DateTime stop = DateTime.Now.AddSeconds(2);
IEnumerable<int> iterable = CountWithTimeLimit(stop);
IEnumerator<int> iterator = iterable.GetEnumerator();

iterator.MoveNext();
Console.WriteLine ("Received {0}", iterator.Current);

iterator.MoveNext();
Console.WriteLine ("Received {0}", iterator.Current);
```

This time the "stopping" line is never printed. It's relatively rare that you'll want to terminate an iterator before it's finished, and it's relatively rare that you'll be iterating manually instead of using `foreach`, but if you *do*, remember to wrap the iterator in a `using` statement.

We've now covered most of the behavior of iterator blocks, but before we end this section it's worth considering a few oddities to do with the current Microsoft implementation.

6.2.4 Quirks in the implementation

If you compile iterator blocks with the Microsoft C#2 compiler and look at the resulting IL in either ildasm or Reflector, you'll see the nested type that the compiler has generated for us behind the scenes. In my case when compiling our (evolved) first iterator block example, it was called `IterationSample.<GetEnumerator>d__0` (where the angle brackets aren't indicating a generic type parameter, by the way). I won't go through exactly what's generated in detail here, but it's worth looking at it in Reflector to get a feel for what's going on, preferably with the language specification next to you: the specification defines different states the type can be in, and this description makes the generated code easier to follow.

Fortunately, as developers we don't need to care much about the hoops the compiler has to jump through. However, there are a few quirks about the implementation that are worth knowing about:

- Before `MoveNext` is called for the first time, the `Current` property will always return `null` (or the default value for the relevant type, for the generic interface).
- After `MoveNext` has returned `false`, the `Current` property will always return the last value returned.
- `Reset` always throws an exception instead of resetting like our manual implementation did. This is required behavior, laid down in the specification.
- The nested class always implements both the generic and nongeneric form of `IEnumerator` (and the generic and nongeneric `IEnumerable` where appropriate).

Failing to implement `Reset` is quite reasonable—the compiler can't reasonably work out what you'd need to do in order to reset the iterator, or even whether it's feasible. Arguably `Reset` shouldn't have been in the `IEnumerator` interface to start with, and I certainly can't remember the last time I called it.

Implementing extra interfaces does no harm either. It's interesting that if your method returns `IEnumerable` you end up with one class implementing five interfaces

(including `IDisposable`). The language specification explains it in detail, but the upshot is that as a developer you don't need to worry.

The behavior of `Current` is odd—in particular, keeping hold of the last item after supposedly moving off it could keep it from being garbage collected. It's possible that this may be fixed in a later release of the C# compiler, though it's unlikely as it could break existing code.[6] Strictly speaking, it's correct from the C#2 language specification point of view—the behavior of the `Current` property is undefined. It would be nicer if it implemented the property in the way that the framework documentation suggests, however, throwing exceptions at appropriate times.

So, there are a few tiny drawbacks from using the autogenerated code, but *sensible* callers won't have any problems—and let's face it, we've saved a *lot* of code in order to come up with the implementation. This means it makes sense to use iterators more widely than we might have done in C#1. Our next section provides some sample code so you can check your understanding of iterator blocks and see how they're useful in real life rather than just in theoretical scenarios.

6.3 Real-life example: iterating over ranges

Have you ever written some code that is really simple in itself but makes your project *much* neater? It happens to me every so often, and it usually makes me happier than it probably ought to—enough to get strange looks from colleagues, anyway. That sort of slightly childish delight is particularly strong when it comes to using a new language feature in a way that is *clearly* nicer and not just doing it for the sake of playing with new toys.

6.3.1 Iterating over the dates in a timetable

While working on a project involving timetables, I came across a few loops, all of which started like this:

```
for (DateTime day = timetable.StartDate;
     day <= timetable.EndDate;
     day = day.AddDays(1))
```

I was working on this area of code quite a lot, and I always hated that loop, but it was only when I was reading the code out loud to another developer as pseudo-code that I realized I was missing a trick. I said something like, "For each day within the timetable." In retrospect, it's obvious that what I really wanted was a `foreach` loop. (This may well have been obvious to you from the start—apologies if this is the case. Fortunately I can't see you looking smug.) The loop is much nicer when rewritten as

```
foreach (DateTime day in timetable.DateRange)
```

In C#1, I might have looked at that as a fond dream but not bothered implementing it: we've seen how messy it is to implement an iterator by hand, and the end result

[6] The Microsoft C#3 compiler shipping with .NET 3.5 behaves in the same way.

only made a few `for` loops neater in this case. In C#2, however, it was easy. Within the class representing the timetable, I simply added a property:

```
public IEnumerable<DateTime> DateRange
{
   get
   {
      for (DateTime day = StartDate;
           day <= EndDate;
           day = day.AddDays(1))
      {
         yield return day;
      }
   }
}
```

Now this has clearly just moved the original loop into the timetable class, but that's OK—it's much nicer for it to be encapsulated there, in a property that *just* loops through the days, yielding them one at a time, than to be in business code that was dealing with those days. If I ever wanted to make it more complex (skipping weekends and public holidays, for instance), I could do it in one place and reap the rewards everywhere.

I thought for a while about making the timetable class implement `IEnumerable<DateTime>` itself, but shied away from it. Either way would have worked, but it so happened that the property led me toward the next step: why should the `DateRange` property just be iterable? Why isn't it a fully fledged object that can be iterated over, asked whether or not it contains a particular date, as well as for its start and end dates? While we're at it, what's so special about `DateTime`? The concept of a range that can be stepped through in a particular way is obvious and applies to many types, but it's still surprisingly absent from the Framework libraries.

For the rest of this section we'll look at implementing a simple `Range` class (and some useful classes derived from it). To keep things simple (and printable), we won't make it as feature-rich as we might want—there's a richer version in my open source miscellaneous utility library[7] that collects odds and ends as I occasionally write small pieces of useful code.

6.3.2 Scoping the Range class

First we'll decide (broadly) what we want the type to do, as well as what it *doesn't* need to be able to do. When developing the class, I applied test-driven development to work out what I wanted. However, the frequent iterative nature of test-driven development (TDD) doesn't work as well in a book as it does in reality, so I'll just lay down the requirements to start with:

- A range is defined by a start value and an end value (of the same type, the "element type").
- We must be able to compare one value of the element type with another.

[7] http://pobox.com/~skeet/csharp/miscutil

- We want to be able to find out whether a particular value is within the range.
- We want to be able to iterate through the range easily.

The last point is obviously the most important one for this chapter, but the others shape the fundamental decisions and ask further questions. In particular, it seems obvious that this should use generics, but should we allow *any* type to be used for the bounds of the range, using an appropriate `IComparer`, or should we only allow types that implement `IComparable<T>`, where `T` is the same type? When we're iterating, how do we move from one value to another? Should we *always* have to be able to iterate over a range, even if we're only interested in the other aspects? Should we be able to have a "reverse" range (in other words, one with a start that is greater than the end, and therefore counts down rather than up)? Should the start and end points be exclusive or inclusive?

All of these are important questions, and the normal answers would promote flexibility and usefulness of the type—but our overriding priority here is to keep things simple. So:

- We'll make comparisons simple by constraining the range's type parameter `T` to implement `IComparable<T>`.
- We'll make the class abstract and require a `GetNextValue` method to be implemented, which will be used during iteration.
- We won't worry about the idea of a range that can't be iterated over.
- We won't allow reverse ranges (so the end value must always be greater than or equal to the start value).
- Start and end points will both be inclusive (so both the start and end points are considered to be members of the range). One consequence of this is that we can't represent an empty range.

The decision to make it an abstract class isn't as limiting as it possibly sounds—it means we'll have derived classes like `Int32Range` and `DateTimeRange` that allow you to specify the "step" to use when iterating. If we ever wanted a more general range, we could always create a derived type that allows the step to be specified as a `Converter` delegate. For the moment, however, let's concentrate on the base type. With all the requirements specified,[8] we're ready to write the code.

6.3.3 Implementation using iterator blocks

With C#2, implementing this (fairly limited) `Range` type is remarkably easy. The hardest part (for me) is remembering how `IComparable<T>.CompareTo` works. The trick I usually use is to remember that if you compare the return value with 0, the result is the same as applying that comparison operator between the two values involved, in the order they're specified. So `x.CompareTo(y) < 0` has the same meaning as `x < y`, for example.

[8] If only real life were as simple as this. We haven't had to get project approval and specification sign-off from a dozen different parties, nor have we had to create a project plan complete with resource requirements. Beautiful!

Listing 6.8 is the complete `Range` class, although we can't quite use it yet as it's still abstract.

Listing 6.8 The abstract `Range` class allowing flexible iteration over its values

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

public abstract class Range<T> : IEnumerable<T>
   where T : IComparable<T>                          ❶ Ensures we can compare values
{
   readonly T start;
   readonly T end;

   public Range(T start, T end)                      ❷ Prevents "reversed" ranges
   {
      if (start.CompareTo(end) > 0)
      {
         throw new ArgumentOutOfRangeException();
      }
      this.start = start;
      this.end = end;
   }

   public T Start
   {
      get { return start; }
   }

   public T End
   {
      get { return end; }
   }

   public bool Contains(T value)
   {
      return value.CompareTo(start) >= 0 &&
             value.CompareTo(end) <= 0;
   }

   public IEnumerator<T> GetEnumerator()             ❸ Implements IEnumerable<T> implicitly
   {
      T value = start;
      while (value.CompareTo(end) < 0)
      {
         yield return value;
         value = GetNextValue(value);
      }
      if (value.CompareTo(end) == 0)
      {
         yield return value;
      }
   }

   IEnumerator IEnumerable.GetEnumerator()           ❹ Implements IEnumerable explicitly
   {
```

```
        return GetEnumerator();
    }

    protected abstract T GetNextValue(T current);    ❺ "Steps" from one value to the next
}
```

The code is quite straightforward, due to C#2's iterator blocks. The type constraint on `T` ❶ ensures that we'll be able to compare two values, and we perform an execution-time check to prevent a range being constructed with a lower bound higher than the upper bound ❷. We still need to work around the problem of implementing both `IEnumerable<T>` and `IEnumerable` by using explicit interface implementation for the nongeneric type ❹ and exposing the generic interface implementation in the more usual, implicit way ❸. If you don't immediately see why this is necessary, look back to the descriptions in sections 2.2.2 and 3.4.3.

The actual iteration merely starts with the lower end of the range, and repeatedly fetches the next value by calling the abstract `GetNextValue` method ❺ until we've reached or exceeded the top of the range. If the last value found is in the range, we yield that as well. Note that the `GetNextValue` method shouldn't need to keep any state—given one value, it should merely return the next one in the range. This is useful as it means that we should be able to make most of the derived types immutable, which is always nice. It's easy to derive from `Range`, and we'll implement the two examples given earlier—a range for dates and times (`DateTimeRange`) and a range for integers (`Int32Range`). They're very short and very similar—listing 6.9 shows both of them together.

Listing 6.9 Two classes derived from `Range`, to iterate over dates/times and integers

```
using System;
public class DateTimeRange : Range<DateTime>
{
    readonly TimeSpan step;

    public DateTimeRange(DateTime start, DateTime end)
        : this (start, end, TimeSpan.FromDays(1))      Uses default step
    { }

    public DateTimeRange(DateTime start,               Uses specified step
                         DateTime end,
                         TimeSpan step)
        : base(start, end)
    {
        this.step = step;
    }

    protected override DateTime GetNextValue(DateTime current)
    {
        return current + step;                         Uses step to find next value
    }
}

public class Int32Range : Range<int>
{
    readonly int step;
```

```
    public Int32Range(int start, int end)
       : this (start, end, 1)
    { }

    public Int32Range(int start, int end, int step)
       : base(start, end)
    {
       this.step = step;
    }

    protected override int GetNextValue(int current)
    {
       return current + step;
    }
}
```

If we could have specified addition (potentially using another type) as a type parameter, we could have used a single type everywhere, which would have been neat. There are other obvious candidates, such as `SingleRange`, `DoubleRange`, and `DecimalRange`, which I haven't shown here. Even though we have to derive an extra class for each type we want to iterate over, the gain over C#1 is still tremendous. Without generics there would have been casts everywhere (and boxing for value types, which probably includes most types you want to use for ranges), and without iterator blocks the code for the separate iterator type we'd have needed would probably have been about as long as the base class itself. It's worth noting that when we use the step to find the next value we don't need to change anything within the instance—both of these types are immutable and so can be freely shared between threads, returned as properties, and used for all kinds of other operations without fear.

With the `DateTimeRange` type in place, I could replace the `DateRange` property in my timetable application, and remove the `StartDate` and `EndDate` properties entirely. The closely related values are now nicely encapsulated, the birds are singing, and all is right with the world. There's a lot more we *could* do to our `Range` type, but for the moment it's served its purpose well.

The `Range` type is just one example of a way in which iteration presents itself as a natural option in C#2 where it would have been significantly less elegant in C#1. I hope you'll consider it next time you find yourself writing a start/end pair of variables or properties. As examples go, however, it's pretty tame—iterating over a range isn't exactly a novel idea. To close the chapter, we'll look at a considerably less conventional use of iterator blocks—this time for the purpose of providing one side of a multithreaded conversation.

6.4 Pseudo-synchronous code with the Concurrency and Coordination Runtime

The *Concurrency and Coordination Runtime* (CCR) is a library developed by Microsoft to offer an alternative way of writing asynchronous code that is amenable to complex coordination. At the time of this writing, it's only available as part of the Microsoft Robotics Studio,[9] although hopefully that will change. We're not going to delve into

the depths of it—fascinating as it is—but it has one very interesting feature that's relevant to this chapter. Rather than present real code that could compile and run (involving pages and pages of description of the library), we'll just look at some pseudo-code and the ideas behind it. The purpose of this section isn't to think too deeply about asynchronous code, but to show how by adding intelligence to the compiler, a different form of programming becomes feasible and reasonably elegant. The CCR uses iterator blocks in an interesting way that takes a certain amount of mental effort to start with. However, once you see the pattern it can lead to a radically different way of thinking about asynchronous execution.

Suppose we're writing a server that needs to handle lots of requests. As part of dealing with those requests, we need to first call a web service to fetch an authentication token, and then use that token to get data from two independent data sources (say a database and another web service). We then process that data and return the result. Each of the fetch stages could take a while—a second, say. The normal two options available are simply synchronous and asynchronous. The pseudo-code for the synchronous version might look something like this:

```
HoldingsValue ComputeTotalStockValue(string user, string password)
{
   Token token = AuthService.Check(user, password);
   Holdings stocks = DbService.GetStockHoldings(token);
   StockRates rates = StockService.GetRates(token);

   return ProcessStocks(stocks, rates);
}
```

That's very simple and easy to understand, but if each request takes a second, the whole operation will take three seconds and tie up a thread for the whole time it's running. If we want to scale up to hundreds of thousands of requests running in parallel, we're in trouble. Now let's consider a fairly simple asynchronous version, which avoids tying up a thread when nothing's happening[10] and uses parallel calls where possible:

```
void StartComputingTotalStockValue(string user, string password)
{
   AuthService.BeginCheck(user, password, AfterAuthCheck, null);
}

void AfterAuthCheck(IAsyncResult result)
{
   Token token = AuthService.EndCheck(result);
   IAsyncResult holdingsAsync = DbService.BeginGetStockHoldings
      (token, null, null);
   StockService.BeginGetRates
      (token, AfterGetRates, holdingsAsync);

}
```

[9] http://www.microsoft.com/robotics

[10] Well, mostly—in order to keep it relatively simple, it might still be inefficient as we'll see in a moment.

```
void AfterGetRates(IAsyncResult result)
{
   IAsyncResult holdingsAsync = (IAsyncResult)result.AsyncState;
   StockRates rates = StockService.EndGetRates(result);
   Holdings holdings = DbService.EndGetStockHoldings
      (holdingsAsync);

   OnRequestComplete(ProcessStocks(stocks, rates));
}
```

This is much harder to read and understand—and that's only a simple version! The coordination of the two parallel calls is only achievable in a simple way because we don't need to pass any other state around, and even so it's not ideal. (It will still block a thread waiting for the database call to complete if the second web service call completes first.) It's far from obvious what's going on, because the code is jumping around different methods so much.

By now you may well be asking yourself where iterators come into the picture. Well, the iterator blocks provided by C#2 effectively allow you to "pause" current execution at certain points of the flow through the block, and then come back to the same place, with the same state. The clever folks designing the CCR realized that that's exactly what's needed for something called a *continuation-passing style* of coding. We need to tell the system that there are certain operations we need to perform—including starting other operations asynchronously—but that we're then happy to wait until the asynchronous operations have finished before we continue. We do this by providing the CCR with an implementation of `IEnumerator<ITask>` (where `ITask` is an interface defined by the CCR). Here's pseudo-code for our request handling in this style:

```
IEnumerator<ITask> ComputeTotalStockValue(string user, string pass)
{
   Token token = null;

   yield return Ccr.ReceiveTask(
      AuthService.CcrCheck(user, pass)
         delegate(Token t){ token = t; }
   );

   Holdings stocks = null;
   StockRates rates = null;
   yield return Ccr.MultipleReceiveTask(
      DbService.CcrGetStockHoldings(token),
      StockService.CcrGetRates(token),
      delegate(Stocks s, StockRates sr)
         { stocks = s; rates = sr; }
   );

   OnRequestComplete(ProcessStocks(stocks, rates));
}
```

Confused? I certainly was when I first saw it—but now I'm somewhat in awe of how neat it is. The CCR will call into our code (with a call to `MoveNext` on the iterator), and we'll execute until and including the first `yield return` statement. The `CcrCheck` method within `AuthService` would kick off an asynchronous request, and the CCR would wait

(without using a dedicated thread) until it had completed, calling the supplied delegate to handle the result. It would then call `MoveNext` again, and our method would continue. This time we kick off two requests in parallel, and the CCR to call another delegate with the results of both operations when they've *both* finished. After that, `MoveNext` is called for a final time and we get to complete the request processing.

Although it's obviously more complicated than the synchronous version, it's still all in one method, it will get executed in the order written, and the method itself can hold the state (in the local variables, which become state in the extra type generated by the compiler). It's fully asynchronous, using as few threads as it can get away with. I haven't shown any error handling, but that's also available in a sensible fashion that forces you to think about the issue at appropriate places.

It all takes a while to get your head around (at least unless you've seen continuation-passing style code before) but the potential benefits in terms of writing correct, scalable code are enormous—and it's only feasible in such a neat way due to C#2's syntactic sugar around iterators and anonymous methods. The CCR hasn't hit the mainstream at the time of writing, but it's possible that it will become another normal part of the development toolkit[11]—and that other novel uses for iterator blocks will be thought up over time. As I said earlier, the point of the section is to open your mind to possible uses of the work that the compiler can do for you beyond just simple iteration.

6.5 Summary

C# supports many patterns indirectly, in terms of it being feasible to implement them in C#. However, relatively few patterns are *directly* supported in terms of language features being specifically targeted at a particular pattern. In C#1, the iterator pattern was directly supported from the point of view of the calling code, but not from the perspective of the collection being iterated over. Writing a correct implementation of `IEnumerable` was time-consuming and error-prone, without being interesting. In C#2 the compiler does all the mundane work for you, building a state machine to cope with the "call-back" nature of iterators.

It should be noted that iterator blocks have one aspect in common with the anonymous methods we saw in chapter 5, even though the actual features are very different. In both cases, extra types may be generated, and a potentially complicated code transformation is applied to the original source. Compare this with C#1 where most of the transformations for syntactic sugar (`lock`, `using`, and `foreach` being the most obvious examples) were quite straightforward. We'll see this trend toward smarter compilation continuing with almost every aspect of C#3.

As well as seeing a real-life example of the use of iterators, we've taken a look at how one particular library has used them in a fairly radical way that has little to do with what comes to mind when we think about iteration over a collection. It's worth bearing in mind that different languages have also looked at this sort of problem

[11] Some aspects of the CCR may also become available as part of the Parallel Extensions library described in chapter 13.

before—in computer science the term *coroutine* is applied to concepts of this nature. Different languages have historically supported them to a greater or lesser extent, with tricks being applicable to simulate them sometimes—for example, Simon Tatham has an excellent article[12] on how even C can express coroutines if you're willing to bend coding standards somewhat. We've seen that C#2 makes coroutines easy to write and use.

Having seen some major and sometimes mind-warping language changes focused around a few key features, our next chapter is a change of pace. It describes a number of small changes that make C#2 more pleasant to work with than its predecessor, learning from the little niggles of the past to produce a language that has fewer rough edges, more scope for dealing with awkward backward-compatibility cases, and a better story around working with generated code. Each feature is relatively straightforward, but there are quite a few of them…

[12] http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

PROGRAMMING LANGUAGES

C# IN DEPTH

Jon Skeet Foreword by Eric Lippert

In programming, there's no substitute for knowing your stuff. In versions 2 and 3, C# introduces new concepts such as lambda expressions and implicit typing that make the language more flexible and give you more power. Using Language INtegrated Query (LINQ)—also new in C# 3—you can interact with data of any type directly from C#. Simply put, mastering these features will make you a more valuable C# developer.

> **"The definitive what, how, and why of C# 3"**
>
> —FROM THE FOREWORD BY Eric Lippert, Microsoft

C# in Depth is designed to bring you to a new level of programming skill. It dives deeply into key C# topics—in particular the new ones. You'll learn to reuse algorithms in a type-safe way with C# 2 generics and expand the functionality of existing classes and interfaces using C# 3 extension methods. Tricky issues become clear in author Jon Skeet's crisp, easy-to-follow explanations and snappy, pragmatic examples. With this book under your belt, you will easily learn—and then master—new frameworks and platforms.

What's Inside

- How and where (and why) to use the new language features
- Backgrounder on C# 1
- Cutting-edge best practices
- Become comfortable and proficient with C# 2 and 3

Jon Skeet is a UK-based developer who is a recognized C# community leader, widely known for clarifying misunderstood aspects of C# and .NET. Jon has been a Microsoft C# MVP since 2003.

For more information and to download code samples visit www.manning.com/CSharpInDepth

"Become a C# 3 maestro!"
—Fabrice Marguerie, C# MVP
author of *LINQ in Action*

"The best C# book I've ever read."
—Chris Mullins, C# MVP

"Clear and concise."
—Robin Shahan, GoldMail.com

"A treat!"
—Anil Radhakrishna
ASP.NET MVP

"Reveals C#'s powerful mysteries."
—Christopher Haupt
BuildingWebApps.com

"So good, it hurts my head."
—J.D. Conley, Hive7 Inc.

"Enriches the beginner, polishes the expert."
—Josh Cronemeyer
ThoughtWorks

ISBN-13: 978-1933988368
ISBN-10: 1933988363
54499
9 781933 988368

MANNING $44.99 / Can $44.99 [INCLUDING EBOOK]