Effects of Yield

In C#, the `yield` keyword is used to generate an implementation of the `Iterator` pattern, which in .NET is an implementation of `IEnumerator`. It is a useful and under-utilized feature of the C# language that can greatly reduce the complexity of iterating lazy or dynamic collections of objects. Methods using the `yield` keyword must return `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, or `IEnumerator<T>`. For type-safety and semantics, it is often preferred to return `IEnumerable<T>`.

What actually happens behind the scenes with a `yield` statement is often hard to understand. The compiler will take your method that uses a `yield` statement and translate it into a class that implements the logic in your method. This class implements the `Iterator` pattern, and each time the iterator is called (the `MoveNext` method), your original method is called again. When the `yield` statement is hit, the `MoveNext` method returns. This means that your method will be called once for each element that will ultimately be returned. The generated code keeps track of state between calls to your method.

This has a couple of advantages. The most important, from a memory perspective, is that you never have to declare the list that the calling code will appear to loop through. Only one item at a time has to be in memory, which can be a dramatic improvement if only one object is used at a time. With the `yield` statement, you can write the logic in a more natural way, as if all of the processing was done in a single method call, and let the compiler handle the state processing across multiple method calls.

Consider the example in Listing 4.49. The processing logic is written in a natural style with a straightforward loop, but we avoid having to ever have all 360 elements in the list in memory at one time. Depending on the size of the elements in the list and the number of elements, this can be a substantial reduction in the memory footprint. Best of all, we don't have to sacrifice readability to get the performance benefits.

```
public IEnumerable <AmortizationScheduleItem> CalculatePayments
    (decimal principal, int term, decimal interestRate,
    decimal monthlyPayment, DateTime startingDate)
{
 decimal totalInterest = 0;
 decimal balanceamount = principal;
 for (int count = 0; count <= term - 1; count++)
 {
   decimal monthInterest = principal*
    ((interestRate/100m)*(1.0m/12.0m));
   decimal monthPrincipal = monthlyPayment - monthInterest;
   totalInterest += monthInterest;
   principal -= monthPrincipal;
   balanceamount = balanceamount - monthPrincipal;
   var monthlyDetail = new AmortizationScheduleItem
         {
           Date = startingDate.AddMonths(count),
           Balance = balanceamount,
           PrincipalPaid= monthPrincipal,
           InterestPaid= monthInterest,
           Payment = monthlyPayment,
           TotalInterest = totalInterest
         };
   yield return monthlyDetail;
 }
```

```
}
```

Listing 4.49: An intuitive payment calculator using `yield` to lower the memory footprint.

The `CalculatePayments` method uses the `yield` statement to return a result as each item is calculated. By calculating all values in a loop, we can easily keep track of intermediate values and use them to build the next result, but behind the scenes we are not running all of these calculations in a single loop. Fortunately, we can generally ignore such pesky details. The compiler takes this simple method and converts it to a class similar to Listing 4.50.

```
[CompilerGenerated]
private sealed class <CalculatePayments>d__5 : IEnumerable<AmortizationScheduleIt
em>, IEnumerable, IEnumerator<AmortizationScheduleItem>, IEnumerator, IDisposable
{
 // Fields
 private int <>1__state;
 private AmortizationScheduleItem <>2__current;
 public decimal <>3__interestRate;
 public decimal <>3__monthlyPayment;
 public decimal <>3__principal;
 public DateTime <>3__startingDate;
 public int <>3__term;
 public Program <>4__this;
 public AmortizationScheduleItem <>g initLocal4;
 private int <>l__initialThreadId;
 public decimal <balanceamount>5__7;
 public int <count>5__8;
 public decimal <monthInterest>5__9;
 public AmortizationScheduleItem <monthlyDetail>5__b;
 public decimal <monthPrincipal>5__a;
 public decimal <totalInterest>5__6;
 public decimal interestRate;
 public decimal monthlyPayment;
 public decimal principal;
 public DateTime startingDate;
 public int term;

 // Methods
 [DebuggerHidden]
 public <CalculatePayments>d__5(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.<totalInterest>5__6 = 0M;
    this.<balanceamount>5__7 = this.principal;
    this.<count>5__8 = 0;
```

```
    while (this.<count>5__8 <= (this.term - 1))
    {
      this.<monthInterest>5__9 = this.principal *
      ((this.interestRate / 100M) * 0.08333M);
      this.<monthPrincipal>5__a = this.monthlyPayment –
       this.<monthInterest>5__9;
      this.<totalInterest>5__6 += this.<monthInterest>5__9;
      this.principal -= this.<monthPrincipal>5__a;
      this.<balanceamount>5 7 -= this.<monthPrincipal>5__a;
      this.<>g__initLocal4 = new AmortizationScheduleItem();
      this.<>g__initLocal4.Date =
       this.startingDate.AddMonths(this.<count>5__8);
      this.<>g__initLocal4.Balance =
       this.<balanceamount>5__7;
      this.<>g__initLocal4.PrincipalPaid=
       this.<monthPrincipal>5__a;
      this.<>g__initLocal4.InterestPaid=
       this.<monthInterest>5__9;
      this.<>g__initLocal4.Payment = this.monthlyPayment;
      this.<>g__initLocal4.TotalInterest =
       this.<totalInterest>5__6;
      this.<monthlyDetail>5__b = this.<>g__initLocal4;
      this.<>2__current = this.<monthlyDetail>5__b;
      this.<>1__state = 1;
      return true;
    Label_0190:
      this.<>1__state = -1;
      this.<count>5__8++;
    }
    break;

   case 1:
    goto Label_0190;
  }
  return false;
}

[DebuggerHidden]
IEnumerator<AmortizationScheduleItem>
   IEnumerable<AmortizationScheduleItem>.GetEnumerator()
{
  Program.<CalculatePayments>d__5 d__;
  if ((Thread.CurrentThread.ManagedThreadId ==
   this.<>l__initialThreadId) && (this.<>1__state == -2))
  {
   this.<>1__state = 0;
   d__ = this;
  }
  else
  {
   d__ = new Program.<CalculatePayments>d__5(0);
   d__.<>4__this = this.<>4__this;
  }
  d__.principal = this.<>3__principal;
```

```
  d__.term = this.<>3__term;
  d__.interestRate = this.<>3__interestRate;
  d__.monthlyPayment = this.<>3__monthlyPayment;
  d__.startingDate = this.<>3__startingDate;
  return d__;
}

[DebuggerHidden]
 IEnumerator IEnumerable.GetEnumerator()
 {
   return this.System.Collections.Generic.
    IEnumerable<ConsoleApplication1.AmortizationScheduleItem>.
     GetEnumerator();
 }

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

void IDisposable.Dispose()
{
}

// Properties
AmortizationScheduleItem IEnumerator<AmortizationScheduleItem>.Current
{
  [DebuggerHidden]
  get
  {
   return this.<>2__current;
  }
}

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

Listing 4.50: The effects of the `yield` expanded by the compiler.

You might be forgiven for thinking *"Wow! The code generated by the compiler is complex and not at all easy to follow*," and that's exactly true. Fortunately, we don't have to write this code or worry about maintaining it. We can write the much easier-to-follow code using the `yield` statement and reap the benefits of the more complicated code that the compiler generates.

Using the code is very intuitive as well. We can treat the method as if it returns the full array and not worry about the magic happening behind the scenes.

```
foreach (var item in a.CalculatePayments(principle ,term,
 interstRate , monthlyPayment , DateTime.Today ))
{
 Console.WriteLine( string.Format( "Month: {0:d}\tBalance: " +
   "{1:c}\tInterest Paid: {2:c}\tPrincipal Paid: " +
   "{3:c}\tTotal Interest Paid: {4:c}",
   item.Date, item.Balance, item.InterestPaid ,
   item.PrincipalPaid, item.TotalInterest ));
}
```

Listing 4.51: Using `CalculatePayments` as if it was a "regular" iterator.

The largest impact the use of the `Iterator` pattern will have on memory and performance is making the decision on whether it is more valuable to lazily retrieve values or retain those values in memory. Lazy retrieval ensures the data is current, keeps the heap clear, but could potentially impact garbage collection with the creation of many small objects. Maintaining the data in memory is better if that data is unlikely to change, or is used more often, but it could move the larger object to a higher generation.

My recommendation is to convert iterators to collection types at application, framework, and layer boundaries. And always test with a profiler to ensure that you get the expected performance results.