The Pragmatic Programmers

# C# Brain Teasers

## Exercise Your Mind

Steve Love

*edited by Adaobi Obi Tulton*

# Deferred Judgment

Defer.cs

```
string[] inputs = [ "128", "256", "<error>", "512" ];

IEnumerable<int> result;
try
{
    result = from number in inputs
             select Convert.ToInt32(number);
}
catch (Exception e)
{
    result = [ 0 ];
}
var count = result.Count();

Console.WriteLine($"There are {count} elements");
```

**Guess the Output**

What’s the output from this program? Try to determine what you expect it to do before moving on.

The output from the program is:

```
<System.FormatException>: The input string '<error>' was not
in a correct format.
```

The exception is caused by attempting to convert "<error>" to an int using the int.Parse static method. Since the string "<error>" cannot be converted to int, a FormatException is thrown, but it's not caught by our catch handler because assigning to result isn't where the int.Parse method is called.

The problem with the code in this puzzle is representative of similar pitfalls that are easily missed by novices and seasoned .NET developers alike. They are all related to the *lazy evaluation* of LINQ expressions.

## Discussion

The reason we get an exception from this code comes down to a single underlying detail: IEnumerable sequences are evaluated *lazily*. Although the expression to initialize the result variable uses the int.Parse method, the method isn't invoked until we call Count.

Many LINQ expressions take advantage of *deferred execution*, sometimes also called *lazy evaluation*. Both of these terms simply mean that some expressions aren't evaluated until they're needed. The result is that some expressions involving LINQ—whether in the query expression format shown here, or its fluent equivalent—don't correspond to the sequential order of the statements and expressions in the code. This isn't a shortcoming of LINQ, it's a feature.

The following shows the fluent equivalent of this puzzle's code:

```
result = inputs.Select(number => Convert.ToInt32(number));
```

The outcome is exactly the same in either case: the result variable does not in fact represent a sequence of int values, it represents only a *potential* sequence of int. This may sound like magic, but it's really just about delegates and coroutines. Oh, and concrete (or crystals, if you're more of a theoretical materials scientist than an applied structural engineer).

## Elements on Demand

A sequence of elements represented by IEnumerable<T> doesn't necessarily contain any elements until those elements are needed. The IEnumerable<T>

variable knows how to produce the real elements, but doesn't store the elements themselves.

There are several LINQ operations that result in an *uncrystallized* sequence where the true value of the elements is not yet known. What's more is that such sequences produce elements on demand.

A Where expression in LINQ, for example, works on one sequence and produces a new sequence by filtering the elements from its input sequence based on a *predicate*—a delegate that returns true or false based on its argument. To illustrate how this works, consider the following contrived example:

```
int [] numbers = [ 20, 5, 2, 1 ];
var result = numbers.Where(n => n == 20);

if (result.Any())
{
    // do something important with small numbers
}
```

The Any method returns true if its input sequence (the result sequence produced by the Where expression) contains anything. To determine that, Any invokes the Where expression, which in turn pulls elements from *its* input sequence (the numbers array) until it finds one that matches its predicate. Here we're looking for numbers that equal 20, so the first element of the array is a match. The remaining elements don't need to be evaluated, and Any can immediately report true.

In this code, the predicate given to Where is evaluated only once. If the numbers array were rearranged so that 20 appeared in last place instead of at the start, then the predicate would need to be evaluated four times: once for each element until a match is found. If there are *no* matching elements, then every number in the numbers array needs to be evaluated in order for Any to return false.

This is where coroutines fit into the picture.

## Coroutines

A *coroutine* is a section of code that can suspend itself and pass control back to its caller. In C# coroutines are used as a convenient way to represent iterators; an *iterator* is an abstract view of a position in a sequence that's independent of both the type of the elements and the sequence itself.

Any method containing the yield keyword is a coroutine and returns an iterator in the form of an IEnumerable<T>. Each time yield is encountered at runtime, the method pauses and control returns to wherever the coroutine method was called. When the coroutine is next invoked, it restarts at the code following the yield statement. Like many other LINQ operations, Where is implemented as a coroutine. The following code shows a simplified view of Where's implementation:

```
IEnumerable<T> Where<T>(this IEnumerable<T> input, Func<T, bool> predicate)
{
    foreach (var element in input)
    {
        if(predicate(element))
            yield return element;
    }
}
```

The loop within Where inspects each element of the input sequence in turn using the predicate delegate. If the predicate never returns true, the input sequence is exhausted and the loop ends. If the predicate returns true for any element, the element is yielded to the calling code and the Where method is suspended. The next time Where is called, it resumes by obtaining the *next* element from the input sequence and continues again until either the predicate returns true or the input sequence is exhausted.

The call to Any in the previous section only invokes Where once, and it returns after inspecting the first element because the predicate n => n == 20 returns true. Since Where isn't invoked again, the remaining elements are never inspected.

## Deferred vs. Concrete Execution

The Any method, along with its counterpart All, are examples of *crystallizing* operations because they cause the sequence to be evaluated, if only partially. Other crystallizing operations including Count and Average need to evaluate the entire sequence to determine their result. Operations like ToArray and ToList make a *concrete* collection from the sequence, and so also cause the whole sequence to be evaluated.

The point of this puzzle is that a sequence isn't evaluated at all until a crystallizing operation is applied to it. The exception caused by the call to int.Parse in the Where method's predicate doesn't get thrown until the offending element ("<error>") is evaluated, and that only happens when result.Count is called—outside of the try…catch block.

Deferred execution is part of the nature of LINQ, and you should try to stay aware of *when* your LINQ expressions get evaluated. This puzzle represents only one of several problems that can result from code that doesn't run in the order in which it's written, but they all follow a similar pattern that you can avoid by learning *when* deferred execution happens—and when it doesn't.

## Further Reading

*Microsoft's documentation has an introduction to LINQ and its syntax at*

https://learn.microsoft.com/en-us/dotnet/csharp/linq/get-started/introduction-to-linq-queries

*LINQ is largely defined by the* Enumerable *type and its extension methods, documented by Microsoft*
*at*

https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable?view=net-9.0

*Jon Skeet describes how coroutines are used to implement iterators in his blog at*

https://www.jonskeet.uk/csharp/csharp2/iterators.html