

Ben Lamb Presents

How to Write Crap Code in C#

Anti-Patterns for
Performance

Wow, so many people interested in writing crap code! (Who'd have thought?)
Given that people usually come to these talks to learn new things I'll assume you're all expert C# coders!

Welcome to Developer Day 4. My name's Ben Lamb.

Today I'm going to tell you **How to Write Crap Code in C# - Anti-Patterns for Performance**.

What We're Not Covering

```
int i = 1;
i = i++;
for (int f = 2; f > i; i--)
{
    Console.WriteLine("Foo");
}
```

This talk is all about writing performant code, not the other kinds of monstrosities that you might regard as crap code.

We'll leave obfuscated code to the Perl programmers, C# isn't very good at it.

Managed languages such as C# are often criticised for being slower than their counterparts such as C++. Often these claims are unfounded. I've worked on several projects where performance has been a problem and often people don't understand why their code is running slowly. Typically they'll blame the .NET runtime as the problem's obvious there rather than in their own code.

So what sort of performance pitfalls await the unwary?

Inspired by the concept of proof by contradiction I decided to take a program and ignore Microsoft best-practises to see "How Slow It Could Go."



I've written a simple application to analyse the complete works of Shakespeare that will allow me to demonstrate various techniques.

The analysers have been written as plug-in modules so can I select one, click Analyse and it returns statistics for the text. All the analysers implement the same functionality, some more quickly than others.

The operation is very simple. Each Shakespeare play is a separate textfile, the program scans all the text files in a folder. Splits the text into lines, removes any punctuation and converts everything to lowercase. Each word is taken in turn and looked up in a master dictionary. If we have already encountered the word we increment the count of the number of times it has occurred, if we haven't seen it before a new entry is created in the dictionary.

That took 2 minutes to run, do you think that's good?

Great Expectations

- What's the limiting factor?
 - Hard disk
 - Memory
 - CPU

How fast should that code run? What's acceptable performance? How much hardware are we expecting to need to run that code when it goes into production?

The complete works of Shakespeare comes to 2Mb. Should it really take 2 minutes to do a simple analysis process 2Mb of data with a 2Ghz CPU!!!

The first thing to ask is what is the limiting factor? We only really have three choices as there's no network involved. The hard disk transfer rate is about 150Mb/sec and its safe to assume that the data would be cached in RAM by the second run anyway. Bus bandwidth is over 1Gb/sec which leaves the CPU.

It's difficult to assess just how quickly the processor should be able to execute some code but Word can grammar check a 2Mb document in less than 2 minutes which is probably a more complicated algorithm than the simple word count I'm doing, this would suggest a problem with my code.

Doing a small test run with a profiler will shed light on the situation and identify performance hotspots but sometimes the problems are at a higher level with the overall design of your code.

Performance Myths

```
int i = 0; // declaring outside the loop is faster
int length = collection.Length; // cache property
    access

// for loop – much quicker than using an
    enumerator
for (i = 0; i < length; i++)
{
    // do something
}
```

Micro optimisations are the little tricks experienced programmers pick up over the years to make their code run faster. Often they take them from one platform to another or pass them on to newer programmers who never verify their effectiveness.

Generally micro optimisations are a waste of time. Use a profiler to work out if they're necessary before applying them. Lots of received wisdom rapidly becomes out-of-date as the CLR is improved. The JIT compiler in the CLR knows lots more about optimisation than you do and has information about lots of special cases.

Why Bother?

Let's buy a faster CPU

Not always possible due to:

- No budget, running costs
- No room in the data centre
- Lead time is too slow

Really crap code will still run like a dog on faster processor

So why bother with performance optimisation at all? Lots of people will tell you to go out and buy a faster machine.

You might find that even faster machine won't make a huge difference. Unless your code is multi-threaded adding more processors won't help and your app might not have been designed to scale over multiple machines. If these two factors are true you can very easily chew up the processing power of the fastest processor on the market by writing crap code.

There may be other reasons why a faster machine isn't an option including: you have no money, you have no room in your data-centre, you need faster performance tomorrow. Remember crap code will still run slowly on a faster processor.

So 2 minutes was rather too slow for that code, let's set a realistic performance benchmark for it. Two seconds, much better! Now let's look at "improving" that.

Flow Control with Exceptions

```
foreach (string word in words)
{
    if (histogram.ContainsKey(word))
    {
        int wordcount = histogram[word];
        wordcount++;
        histogram[word] = wordcount;
    }
    else
    {
        histogram.Add(word, 1);
    }
}
```

My first anti-pattern is using exceptions for flow control. This one's very simple. My analyser has a tight inner loop that iterates over every word in each text file.

Flow Control with Exceptions (2)

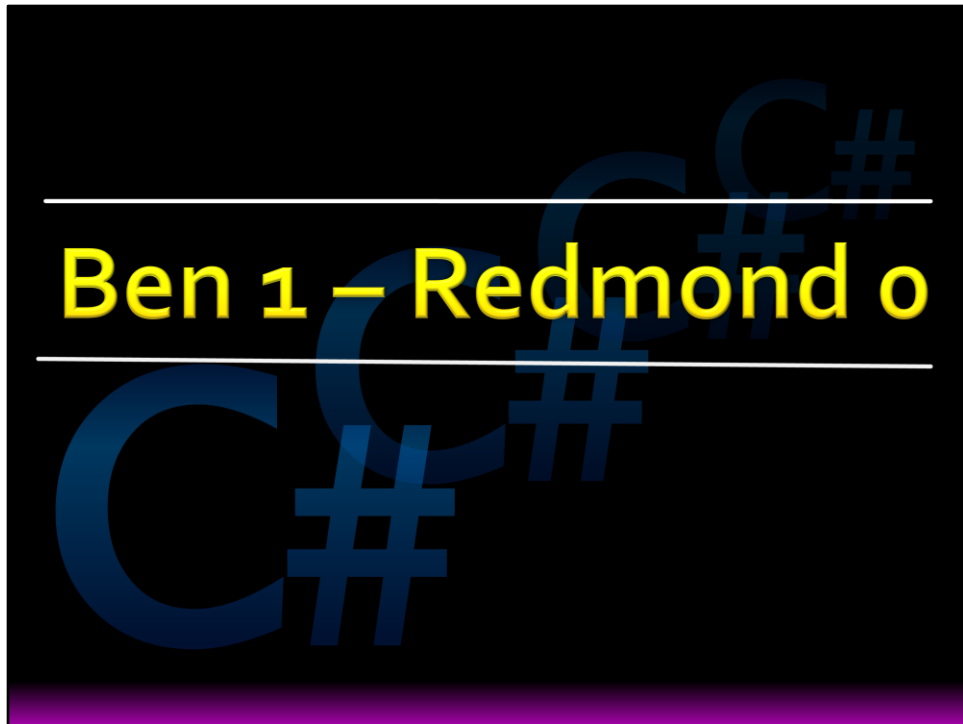
```
foreach (string word in words)
{
    try
    {
        histogram.Add(word, 1);
    }
    catch (ArgumentException)
    {
        int wordcount = histogram[word];
        wordcount++;
        histogram[word] = wordcount;
    }
}
```

Rather than check to see whether the word already exists in the dictionary I'm going to add it regardless and catch the **InvalidArgumentException** that's thrown if the key already exists.



DEMO: Flow Control With Exceptions

A good way of analysing existing code for this anti-pattern is to use the **Reliability and Performance Monitor**, that's the tool formerly known as **Perfmon** if you're not running Windows Vista. Add the counters for **# Exceptions Thrown** and **# Exceptions Thrown/sec**.



Excellent, a score of nearly one minute. Ben 1, Redmond nil! That was a really extreme example of how not to use exceptions, not only did I know that the exceptional occurrence would occur frequently I was using it in a tight inner loop in my code. The runtime really stood no chance. Remember, exceptions are there to catch the unexpected.

This example runs, much, much slower in debug mode. Making your programs hard to debug is not a good idea.

One disclaimer – your mileage may vary. Benchmarks can be used to prove anything the designers intended. You'll find examples on the net showing the exceptions can be really quick. I'd suggest that in real code these really quick benchmarks don't apply.

Abusing Threads

- Threads can be created by:
 - Using asynchronous callbacks
 - Using the ThreadPool
 - Manually using new Thread()

The second anti-pattern is abusing threads.

Since each Shakespeare play is in a separate text file I thought it would be very easy to parallelise my program by making it multi-threaded. The observant amongst you might have noticed that I'm already running the analysis code in a separate thread, this is standard practise to ensure the GUI remains responsive while other processing takes place.

There are three ways to use threads in C#. The first method, an asynchronous callback, is great for situations like the one I just described where you want to use a thread to perform a long-running background operation. Once you've started the operation you don't really have any control over that thread.

The thread pool is for situations where you require multiple threads. The great thing about it is that recycles threads rather than creating new ones every time and it places a limit on the number of threads in existence.

But real crap code aficionados will attempt to hammer CPU by creating large numbers of threads regardless of how many processor cores are present. Unfortunately this doesn't work as context switching between threads is a relatively cheap operation.

To really achieve poor performance with multiple threads you'll need to do some locking, preferably on a single object instance used by all threads. I thought my global dictionary would be ideal.

This is how I adapted my program to abuse threads.

Abusing Threads (2)

```
public override void StartAnalysis(string directory)
{
    List<string> files = ScanDirectory(directory);

    foreach (string filename in files)
    {
        StartThreadToAnalyseFile(filename);
    }

    foreach (Thread thread in threads)
    {
        thread.Join();
    }
}
```

As usual the program scans the directory structure and returns a list of files. It then start a thread to analyse each one and then waits for all the threads to complete.

Abusing Threads (3)

```
private void
  StartThreadToAnalyseFile(string filename)
  {
    Thread thread = new Thread(AnalyseFile);
    threads.Add(thread);
    thread.Start(filename);
  }
```

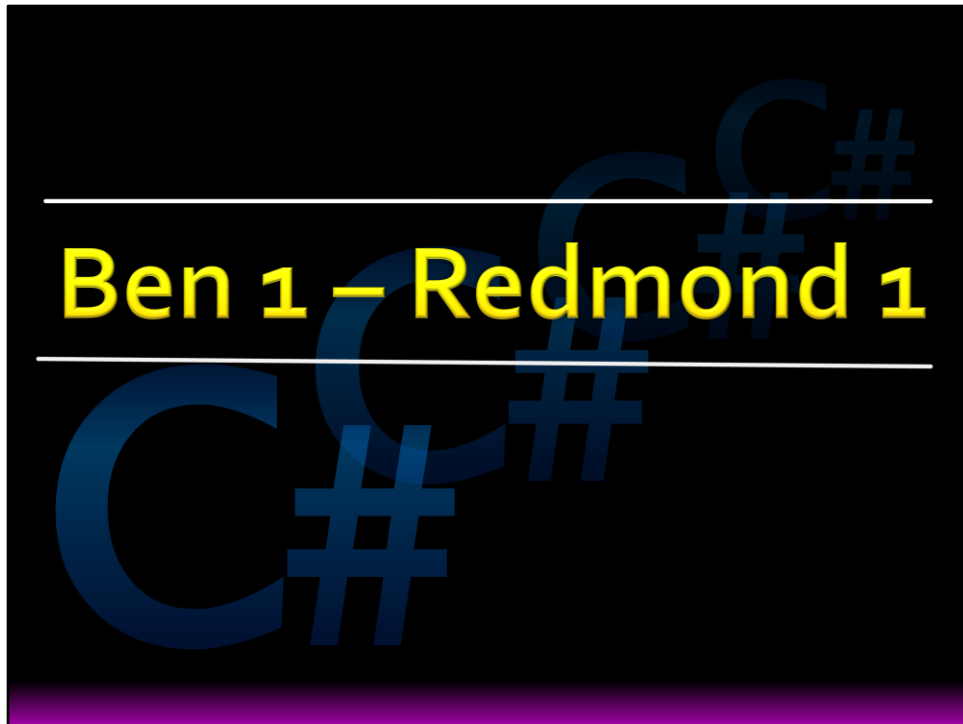
This is the code to create a thread, the Thread constructor takes a single parameter which is a delegate that takes an object and returns a void. The AnalyseFile method matches this delegate. When the thread is started the filename to analyse is passed through.

The thread is added to a list so that we can keep track of them.



DEMO: Abuse of Threads

How Slow Can it Go? – Let's find out...



What a disappointment! Only two and a half seconds. What went wrong?

Well, despite the lock on the dictionary taken out by each thread. It wasn't being held for very long so none of the threads really had to wait around. The actual process of obtaining and relinquishing the lock wasn't nearly as expensive as I thought. I think all I've done in this case is ensured that the code wouldn't show any performance gains if I ran it on multiple processors.

A few years ago my advice would have been to use threads in GUIs where you need background processing in order to keep the GUI responsive. And also in situations where you needed to handle concurrent requests e.g. on a webserver. For batch processing single-threaded apps were the most efficient way to go. However, with multi-core processors becoming the norm and dual-CPU's as cheap as chips it makes sense to parallelise your applications as much as possible, just avoid locking.

Misuse of the Heap

- Avoid boxing/unboxing
- Use generic classes
- Cache locality
- Prefer code that works to fast code

Anti-pattern number three is misusing the heap.

The CLR makes a distinction between value types, which live on the stack, and objects which live on the heap. Value types cannot be referenced by other objects so the CLR has to convert them into objects, this is known as boxing. It involves copying the data from the stack to the heap and adding some additional metadata. Most collections hold references to objects so a classic anti-pattern was to add lots of value types to an ArrayList. Alas generics has taken that fun away from us so we'll have to try something new.

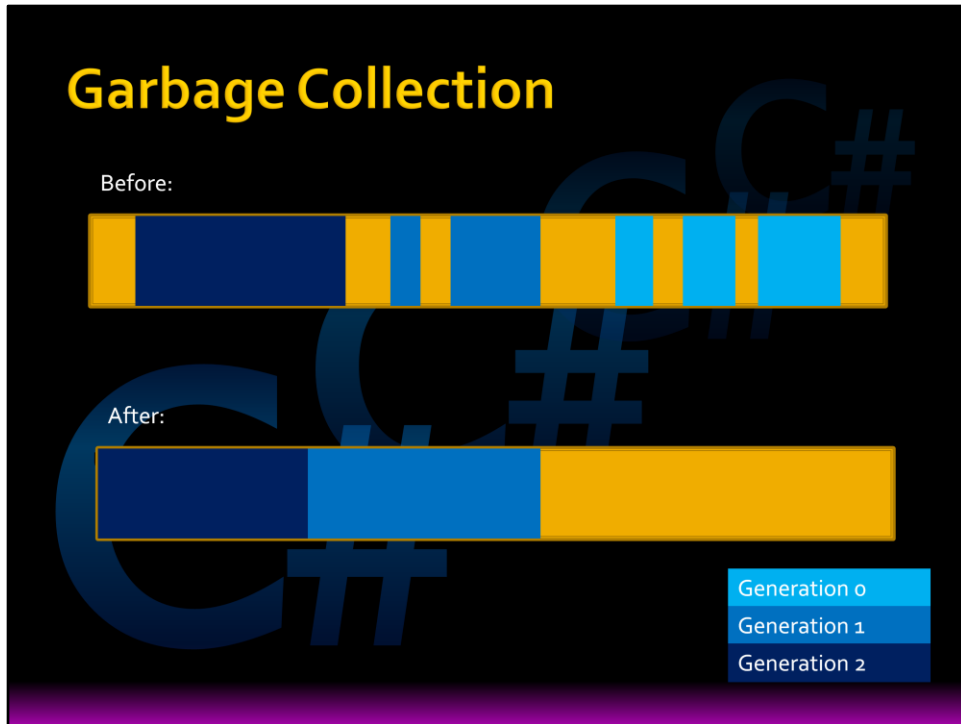
Where possible always use the generic classes, they're faster as lots of boxing/casting operations disappear and more importantly you get type safety.

Caching is another thing to bear in mind. The Front Side Bus (FSB) linking your processor to your memory typically runs at 500ish Mhz, much slower than your 2Ghz processor. Double data rates and other trickery aside it's still much slower. Ideally as much data as possible should be in the processor's level 1 cache, which is about 64Kb. The secondary cache might be 1-2Mb. Structs are smaller than objects and if you have an array of them they'll be contiguously stored in memory which is optimal from a caching point of view when you're doing operations on the array.

The best general advice is to always keep your objects as small as possible by not storing extraneous data.

On that point, given the choice between correct code and fast code I'll choose correct code any day. Don't bother disabling bounds checking on arrays for the sake of performance, it's there for a reason – to ensure your code works.

Garbage Collection



One of the nice things about .NET is the garbage collector, however, not everyone sees it that way. The purpose of the garbage collector is to free up memory that has been allocated to objects that are no longer in use.

One way of implementing garbage collection is **Reference Counting**. The runtime maintains a counter on each object that records the number of references to it. The counter has to be locked prior to being adjusted so that the object can be shared between threads.

One of my C++ programming colleagues was bemoaning the speed of the C# garbage collector and accusing reference counting of being the problem. I pointed out that 1) locking isn't that expensive under C# as we've already seen and 2) the C# garbage collector doesn't use any reference counting whatsoever!!!

This was also a case where my colleague hadn't bothered to do any measuring and had just plucked something out of thin air to blame for the problem.

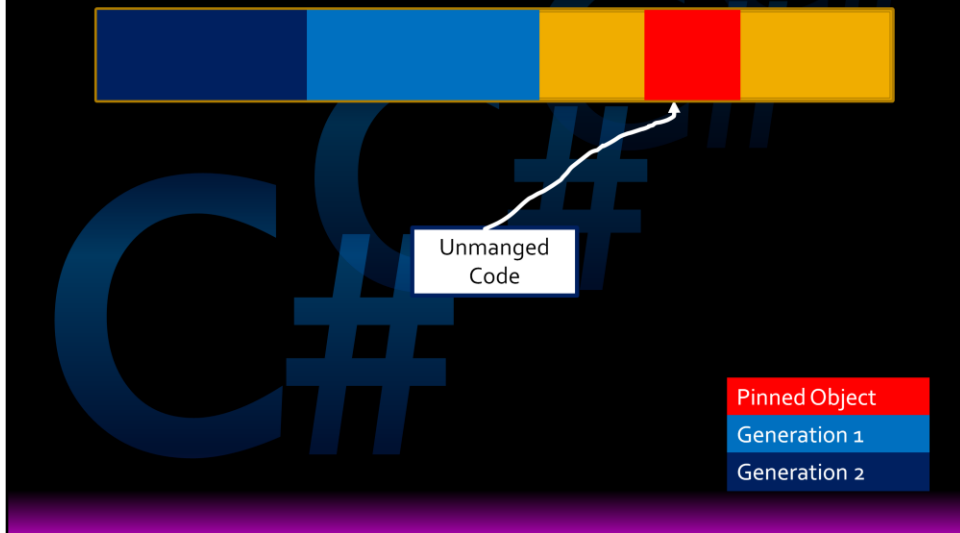
Actually: there wasn't a problem, he was just trying to have a dig at C#.

The .NET implementation of garbage collection uses an algorithm known as **mark and sweep**. After a certain amount of memory has been allocated the garbage collector will intervene and trace from the root objects all referenced objects and mark them. Anything left unmarked is swept up i.e. disposed of. To make allocation of new memory faster all the remaining objects are then moved so there are no gaps between them, this is known as compacting the heap. This means that when it needs to allocate memory the runtime just goes straight to the top of the heap rather than searching the through the heap for a block of the required size. It also stops the heap from becoming riddled with sections of unallocated memory that are too small to accommodate anything useful.

Every time an object survives a garbage collection its generation count is incremented. The idea is that the garbage collector can save time by only looking at the youngest generation of objects on most occasions. These will be found towards the top of the heap and are short lived.

There are three generations of objects in .NET, at various intervals the garbage collector will do a full sweep that includes generations 1 and 2.

Pinned Objects



Things are slightly complicated by the concept of pinned objects. Any object marked as pinned cannot be moved as its address in memory has to remain fixed. This situation typically arises when some unmanaged code is holding a reference to a .NET object. The garbage collector is not capable of updating the reference held by the unmanaged code so cannot move the object.

The garbage collector works extremely well and most of the time you won't notice it's there. C++ fans will sneer that it's not deterministic but on the desktop not a lot is.

Obviously if your program spends too much time in the garbage collector it will run very slowly. The way to achieve this is to ensure that the GC has to do lots of full collections of both generation 1 and 2 objects. This will take much longer than just clearing up the short-lived generation 0 objects.

Fun With Strings

```
public static string NormaliseLine(char[] input)
{
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < input.Length; i++)
    {
        char x = input[i];
        if (Char.IsLetter(x) ||
            Char.IsSeparator(x))
        {
            sb.Append(x);
        }
    }
    return sb.ToString().ToLower();
}
```

For this example I've modified the function that removes punctuation from the text to use strings rather than a `StringBuilder`.

Strings are immutable objects, i.e. they can't be changed, therefore every time we make changes to them the runtime has to allocate a new string and copy of the contents of the old one to it. Remember the garbage collector kicks in according to volume of allocation.

Fun With Strings

```
public static string NormaliseLineSlowVariant(char[] input)
{
    string sb = String.Empty;
    for (int i = 0; i < input.Length; i++)
    {
        char x = input[i];
        if (Char.IsLetter(x) ||
            Char.IsSeparator(x))
        {
            sb += x;
        }
    }
    return sb.ToLower();
}
```

So, for the final time we ask the question...

Timing

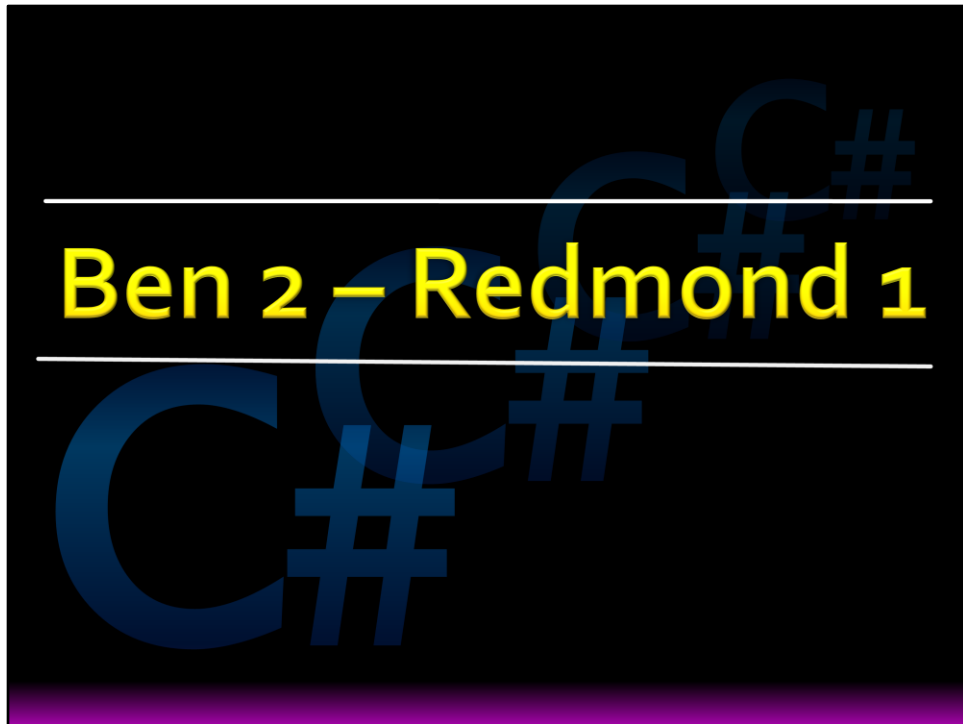
- A: Use Windows high resolution timers
- B: `System.Diagnostics.Stopwatch`
- C: Other

I haven't said anything about how I'm timing my analysers. So I'd like to ask whether you think I'm A) Using Windows high resolution timers, B) The very useful .NET class `System.Diagnostics.Stopwatch` or C) Something else.

`System.Diagnostics.Stopwatch` will use Windows high resolution timers behind the scenes as long as you're not using a legacy platform such as Windows 98. I decided to go for C, an animated clock rendered using Windows Presentation Foundation.



Let's find out!



One minute, what a brilliant result! I actually had to carefully tune that last demo as it was easily taking over ten minutes to run.

It was a bit extreme but shows how a very minor code change can have a big impact. Artificial example? The suggestion to use string concatenation actually came from a guy who wrote a C# compiler and found that was the cause of its sluggishness.

Thanks Migel.

Takeaways

- Managed code is great but you have to know what's going on under the hood.
- Measure it, measure it, measure it!
- The performance of your code matters.

I've demonstrated a number of techniques that can significantly reduce the speed of your code. The time taken to process a 2Mb dataset increased from 2 seconds to nearly a minute. If we were analysing Project Gutenberg which weighs in at 150Gb of text, that's a difference between 17 hours vs 52 days.

Managed code isn't necessarily slow but you have to know what the runtime is doing behind the scenes to be able to write performant code and optimise existing code. You have to think about how quickly you expect your code to run at the start of your project so can make the appropriate design decisions and won't get caught out when someone accuses your code of being slow.

Measure your code before making any optimisations – don't make random guesses.