## Conclusion

As we have seen, memory management touches on every part of the .NET framework; make no mistake, we've covered a good deal of material here. To start with, we've reviewed the Type system and explored the implications of the `Disposable` pattern, and we've also reviewed some of the troubles that you can run into with string manipulation, and the implications of the fact that strings are immutable.

You should now be familiar with the differences between a class and a struct, and how these differences can impact your memory footprint. We've also worked through how to properly implement a struct so that it will be well behaved for hashing, as well as how to avoid some of the potential problems of boxing a struct.

We traced through the history of elevating functions to fully-fledged data types, reviewed the evolution from delegates to anonymous functions to lambdas, and then reviewed closures and how they can extend the lifespan of referenced objects.

We also saw how the new `yield` statement can impact our memory footprint; using this new keyword, we can process a collection of data without having to load the entire collection into memory. This allows us to keep our code well structured, but still take advantage of reduced memory requirements.

Along the way, we saw how handy data structures such as the generic list make it easy to manipulate and process lists of data. This data type is so simple to use that it is easy to forget about what is happening in the background.

We explored some of the implications of not properly initializing the size of the `List <T>` with the constructor (if you're not careful, you can potentially have lists that are sized for twice as much data as they actually hold!).

Having made it through this *tour de force* of the .NET framework, we are ready to move on to the next chapter and see how these concepts contribute to some recurring problems in various .NET applications and frameworks. Take a deep breath, and let's jump right back in, in Chapter 5.